Perl XS - Tutorial

Steven Schubiger

17. Juni 2008

Zusammenfassung

Dieses Tutorial wird den geneigten Leser in den Schritten unterweisen, die nötig sind, um eine eigene compilierte Perl Erweiterung zu erstellen.

Das Tutorial beginnt mit sehr simplen Beispielen und wird mit jedem Beispiel, welches neue Optionen zum Repertoire hinzufügt, komplexer. Gewisse Konzepte mögen nicht ausreichend erläutert sein bis zu einem späteren Zeitpunkt mit der Absicht den Leser sachte in die Thematik einzuweihen.

Dieses Tutorial wurde mit dem Fokus auf Unix geschrieben. Sofern es gewisse Unterschiede zwischen Plattformen geben sollte, werden jene aufgelistet. Sollte eine Abweichung nicht dokumentiert sein, bittet der Autor um eine Benachrichtigung.

Inhaltsverzeichnis

1 Notizen am Rande

1.1 make

Diese Einführung nimmt an, dass das make Programm, welches Perl verwendet, `make` lautet. Anstatt make in den Beispielen, die folgen werden, zu verwenden,

möge man das make spezifizieren, welches Perl gemäss Konfiguration verwendet. Jene Information ist durch **perl -V:make** in Erfahrung zu bringen.

1.2 Versionen

Wenn man eine Perl Erweiterung für den generellen Gebrauch verfasst, ist zu erwarten, dass sie mit Perl Versionen verwendet wird, die von der eigenen Version abweichen. Da du dieses Dokument liest, ist es wahrscheinlich, dass deine Versionnummer von Perl 5.005 oder neuer ist; deine User mögen jedoch vielleicht ältere Versionen haben.

Um zu verstehen was für Inkompatiblitäten man erwarten könnte, und in dem seltenen Falle, dass deine Perl Version älter wie dieses Dokument ist, siehe die Sektion "Fehlerbehebung in den Beispielen".

Sofern deine Erweiterung gewisse Optionen von Perl gebraucht, welche in älteren Versionen nicht verfügbar sind, würden deine Benutzer eine frühe sinnvolle Warnung begrüssen. Du solltest jene wahrscheinlich in die *README* Datei packen, aber heutzutage werden Erweiterungen z.T. automatisch installiert, geführt durch das *CPAN.pm* Modul oder anderen Modulen.

In MakeMaker-basierenden Installationen, stellt das *Makefile.PL* die früheste Möglichkeit dar, um Versionsprüfungen durchzuführen. Man kann zB. für jenen Zweck das *Makefile.PL* so aussehen lassen:

```
eval { require 5.007 }
    or die <<EOD;
############
### Dieses Modul gebraucht das frobnication Framework,
### welches nicht verfuegbar ist vor der Perl Version
### 5.007. Erneuern Sie ihr Perl bevor Sie Kara::Mba
### installieren.
############
EOD
```

1.3 Dynamisches laden versus statisches laden

Es ist eine von vielen geteilte Ansicht, dass wenn man nicht die Möglichkeit hat dynamisch eine Bibliothek zu laden, man keine XSUBs compilieren kann. Dies ist nicht wahr. Man *kann* sie compilieren, aber man muss die XSUBs Routinen mit dem restlichen Perl linken, um somit eine ausführbare Datei zu erzeugen. Ähnlich ist es in Perl 4.

Selbst auf solch einem System kann diese Einführung nutzbringend verwendet werden. Der XSUB Erzeugungsmechanismus wird das System überprüfen und nach Möglichkeit eine dynamisch-ladbare Bibliothek erzeugen; wenn nicht

möglich, wird eine statische Bibliothek ins Leben gerufen und eine zugehörige, ausführbare Datei mit jener Bibliothek gelinkt.

Sollte es erwünscht sein eine statisch-gelinkte, ausführbare Datei auf einem System zu erzeugen, welches dynamisches-laden erlaubt, möge man in den folgenden Beispielen, wo `make` ohne Argumente ausgeführt wird, stattdessen den Befehl `make perl` ausführen.

2 Einführung

2.1 Beispiel 1

Unsere erste Erweiterung wird überaus einfach sein. Wenn wir die Routine in der Erweiterung aufrufen, wird sie eine wohlbekannte Nachricht ausgeben und zurückkehren.

Führe `h2xs -A -n Mytest` aus. Dies wird ein Verzeichnis namens Mytest erstellen, möglicherweise unter ext/ wenn jenes Verzeichnis im aktuellen existiert. Mehrere Dateien werden im Mytest Verzeichnis erzeugt, u.a. MANIFEST, Makefile.PL, Mytest.pm, Mytest.xs, test.pl und Changes.

Die MANIFEST Datei enthält die Namen aller Dateien, die soeben im Mytest Verzeichnis erzeugt wurden.

Die Datei Makefile.PL sollte in etwa so aussehen:

```
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'
);
```

Die Datei Mytest.pm sollte in etwa so starten:

```
package Mytest;

use strict;
use warnings;

require Exporter;
require DynaLoader;
```

```
our @ISA = qw(Exporter DynaLoader);
# Items to export into callers namespace by default. Note: do not export
# names by default without a very good reason. Use EXPORT_OK instead.
# Do not simply export all your public functions/methods/constants.
our @EXPORT = qw(

);
our $VERSION = '0.01';

bootstrap Mytest $VERSION;

# Preloaded methods go here.

# Autoload methods go after __END__, and are processed by the autosplit program.

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

Der Rest der .pm Datei enthält Beispielcode um Dokumentation zur Verfügung zu stellen.

Die Mytest.xs Datei sollte schlussendlich etwa so aussehen:

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

MODULE = Mytest    PACKAGE = Mytest
```

Man möge die .xs Datei editieren, um jenes am Ende hinzuzufügen:

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

Es ist annehmbar das die Zeilen beginnend bei CODE: nicht eingerückt werden. Wie dem auch sei, die Lesbarkeit betreffend, wird es empfohlen, dass man CODE: eine Stufe einrückt und die nachfolgenden Zeilen noch eine.

Führe nun `perl Makefile.PL` aus. Dies wird ein reelles Makefile erzeugen, welches make benötigt. Die Ausgabe ist in etwa so:

```
 % perl Makefile.PL
 Checking if your kit is complete...
```

```
Looks good
Writing Makefile for Mytest
%
```

Nun, make ausführend wird in etwa solch eine Ausgabe produzieren (gewisse Zeilen wurden der Leserlichkeit halber gekürzt und gewisse andere wurden entfernt):

```
% make
umask 0 && cp Mytest.pm ./blib/Mytest.pm
perl xsubpp -typemap typemap Mytest.xs >Mytest.tc && mv Mytest.tc Mytest.c
Please specify prototyping behavior for Mytest.xs (see perlxs manual)
cc -c Mytest.c
Running Mkbootstrap for Mytest ()
chmod 644 Mytest.bs
LD_RUN_PATH="" ld -o ./blib/PA-RISC1.1/auto/Mytest/Mytest.sl -b Mytest.o
chmod 755 ./blib/PA-RISC1.1/auto/Mytest/Mytest.sl
cp Mytest.bs ./blib/PA-RISC1.1/auto/Mytest/Mytest.bs
chmod 644 ./blib/PA-RISC1.1/auto/Mytest/Mytest.bs
Manifying ./blib/man3/Mytest.3
%
```

Man kann die Zeile über 'prototyping behavior' ohne weitere Bedenken ignorieren.

Wenn du über ein Win32 System verfügst und der Compilierprozess scheitert mit Linker Fehlermeldungen für Funktionen in der C Bibliothek, überprüfe ob dein Perl konfiguriert wurde, um PerlCRT zu gebrauchen (**perl -V:libc** ausführend sollte jenes anzeigen, sofern es der Fall sein sollte). Wenn dein Perl konfiguriert wurde um PerlCRT zu gebrauchen, vergewissere dich, ob die PerlCRT.lib an der selbigen Stelle wie msvcrt.lib vorhanden ist, sodass der Compiler jene selber orten kann. Die msvcrt.lib wird gewöhnlicherweise in dem lib Verzeichnis von dem Visual C Compiler gefunden (d.h. C:/DevStudio/VC/lib).

Perl verfügt über eine spezielle Möglichkeit einfach Test Scripts zu verfassen, aber nur für dieses Beispiel werden wir unser eigenes Test Script erzeugen. Kreiere eine Datei namens hello, welche wie folgt aussieht:

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

use ExtUtils::testlib;

use Mytest;

Mytest::hello();
```

Nun setzen wir die Dateirechte des Scripts auf ausführbar (`chmod +x hello`), führen es aus und sehen folgende Ausgabe:

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

2.2 Beispiel 2

Nun werden wir unserer Erweiterung eine Subroutine hinzufügen, welche ein einzelnes numerisches Argument als Eingabe entgegenimmt und wenn die Nummer durch zwei teilbar ist, 0 retourniert, ansonsten 1.

Füge folgendes am Ende von Mytest.xs an:

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

Es braucht kein Leerzeichen am Anfang der Zeile `'int input'` zu sein, es erhöht jedoch die Lesbarkeit. Ob ein Semikolon am Ende jener Zeile verwendet wird ist optional. Eine variable Anzahl an Leerzeichen kann zwischen `int` und `input` eingefügt werden.

Führe nochmals make aus, um die neue geteilte Bibliothek zu compilieren.

Nun führe nochmals die gleichen Schritte aus wie zuvor, ein Makefile von der Makefile.PL Datei erzeugend und make ausführend.

Um so sehen, ob unsere Erweiterung funktioniert, müssen wir die Datei test.pl begutachten. Diese Datei ist so strukturiert, dass sie dieselben Test Strukturen, welche Perl hat, imitiert. Innerhalb des Test Scripts, werden eine Anzahl an Tests ausgeführt, um die Funktionalität zu bestätigen. 'ok' ausgebend wenn der Test korrekt ist, 'not' wenn nicht. Ändere die print Anweisung in dem BEGIN Block, sodass sie 1..4 ausgibt und füge folgenden Code am Ende hinzu:

```
print &Mytest::is_even(0) == 1 ? "ok 2" : "not ok 2", "\n";
print &Mytest::is_even(1) == 0 ? "ok 3" : "not ok 3", "\n";
print &Mytest::is_even(2) == 1 ? "ok 4" : "not ok 4", "\n";
```

Wir werden jenes Test Script durch den Befehl `make test` ausführen. Du solltest in etwa folgende Ausgabe sehen:

```
% make test
PERL_DL_NONLAZY=1 /opt/perl5.004/bin/perl (lots of -I arguments) test.pl
```

```
1..4
ok 1
ok 2
ok 3
ok 4
%
```

2.3 Was geschah?

Das Programm h2xs ist der Ausgangspunkt, um Erweiterungen zu kreieren. In späteren Beispielen werden wir sehen, wie man h2xs gebrauchen kann, um Header Dateien zu lesen und Vorlagen zu erzeugen, um C Routinen aufzurufen.

h2xs erzeugt eine Anzahl an Dateien im Erweiterungs Verzeichnis. Die Datei Makefile.PL ist ein Perl Script, welches ein echtes Makefile generiert um die Erweiterung zu 'compilieren'. Wir werden später einen genaueren Blick drauf werfen.

Die .pm und .xs Dateien sind die Grundlage der Erweiterung. Die .xs Datei beinhaltet die C Routinen, welche die Erweiterung ausmachen. Die .pm Datei enthält Routinen, die Perl das laden der Erweiterung ermöglicht.

Die Generierung des Makefile (`make` ausführend) kreierte ein Verzeichnis namens blib (Synonym für build library) in dem aktuellen Verzeichnis. Dieses Verzeichnis beinhaltet die geteilte Bibliothek, welche wir 'compilieren' werden. Wenn wir sie getestet haben, können wir sie an der finalen Stelle installieren.

Das Test Script via `make test` ausführend tat etwas sehr wichtiges. Es rufte Perl mit all jenen -I Argumenten auf, sodass es die verschiedenen Dateien finden konnte, die Teil der Erweiterung sind. Es ist sehr wichtig, dass wenn du Erweiterungen testest, `make test` verwendest. Wenn du versuchst das Test Script nur für sich selbst auzurufen, wirst du einen fatal error erzeugen. Ein anderer wichtiger Grund um `make test` zu gebrauchen, ist wenn du ein Upgrade einer bestehenden Erweiterung testest, wird `make test` die neue Version der Erweiterung gebrauchen.

Wenn Perl eine `use extension;` sieht, sucht es nach einer Datei mit demselben Namen wie der `used` extension mit einem .pm Suffix. Wenn jene Datei nicht gefunden werden kann, gibt Perl einen fatal error aus. Der Standard Suchpfad ist im `@INC` Array enthalten.

In unserem Falle, teilt Mytest.pm Perl mit, dass es die Exporter und Dynamic Loader Erweiterungen benötigt. Nachfolgend setzt es das `@ISA` und `@EXPORT` Array und den `$VERSION` String; abschliessend teilt es Perl mit, dass es das Modul laden (bootstrapen) soll.

Die beiden Arrays `@ISA` und `@EXPORT` sind sehr wichtig. Das `@ISA` Array enthält eine Liste anderer Pakete in welchen man nach Methoden zu suchen hat, die im aktuellen Paket nicht existieren. Dies ist in der Regel nur von Bedeutung für objekt-orientierte Erweiterungen; normalerweise muss es nicht modifiziert werden.

Das `@EXPORT` Array teilt Perl mit welche der Variablen und Subroutinen der Erweiterung im Namensraum des rufenden Pakets platziert werden sollen. Da wir nicht wissen, ob der Benutzer bereits unsere Variabeln und Subroutinen Namen verwendet, ist es wesentlich sorgfältig zu selektieren, was exportiert werden soll. Exportiere nicht Methoden oder Variabeln automatisch ohne einen guten Grund.

Als eine Faustregel, wenn das Modul objekt-orientiert zu sein versucht, exportiere nichts. Wenn es nur eine Sammlung an Funktionen und Variabeln ist, kannst du via einem anderen Array, dem sog. @EXPORT_OK exportieren. Dieses Array platziert nicht automatisch Subroutinen und Variabelnamen im Namensraum ausser der Benutzer fordert es explizit so an.

Die `$VERSION` Variable wird gebraucht um sicherzustellen, dass die .pm Datei und die geteilte Bibliothek 'in sync' sind. Nach jeder Änderung, sollte man den Wert jener Variable inkrementieren.

2.4 Gute Tests schreiben

Die Wichtigkeit gute Test Scripts zu schreiben kann nicht übermässig betont werden. Man sollte nah am Geschehen dem 'ok/not ok' Stil folgen, welcher Perl gebraucht, sodass es sehr einfach und nicht zweideutig ist das Resultat singemäss zu interpretieren. Sofern du einen Fehler findest und jenen behebst, schreibe einen Test dafür.

Durch Ausführung von `make test`, stellst du sicher, dass dein test.pl läuft und das es die korrekte Version deiner Erweiterung gebraucht. Wenn du viele Testfälle haben solltest, ziehe in Betracht Perl's Test Stil zu verwenden. Erstelle ein Verzeichnis namens t im Erweiterungs Verzeichnis und fügen das Suffix .t den Namen deiner Dateien an. Wenn du `make test` eingibst, werden all jene Dateien ausgeführt.

2.5 Beispiel 3

Unsere dritte Erweiterung wird ein Argument als Eingabe nehmen, jenes runden und den gerundeten Wert dem *Argument* zuweisen.

Füge folgendes dem Ende von Mytest.xs an:

```
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
```

Editiere das Makefile.PL, sodass die korrespondierende Zeile wie folgt aussieht:

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

Generiere das Makefile und führe make aus. Ändere den BEGIN Block, sodass er 1..9 ausgibt und füge folgendes dem test.pl hinzu:

```
$i = -1.5; &Mytest::round($i); print $i == -2.0 ? "ok 5" : "not ok 5", "\n";
$i = -1.1; &Mytest::round($i); print $i == -1.0 ? "ok 6" : "not ok 6", "\n";
$i = 0.0; &Mytest::round($i); print $i == 0.0 ? "ok 7" : "not ok 7", "\n";
$i = 0.5; &Mytest::round($i); print $i == 1.0 ? "ok 8" : "not ok 8", "\n";
$i = 1.2; &Mytest::round($i); print $i == 1.0 ? "ok 9" : "not ok 9", "\n";
```

`make test` ausführend sollte für alle neun Tests ok ausgeben.

Nehme zur Kenntnis, dass in diesen neuen Testfällen eine skalare Variable übergeben wurde. Du wirst dich vielleicht wundern, ob man eine Konstante oder Literale runden kann. Um zu sehen, was passiert, füge temporär folgende Zeile test.pl hinzu:

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

Führe `make test` aus und stelle fest, dass Perl einen fatal error ausgibt. Perl lässt dich nicht den Wert einer Konstante ändern!

2.6 Was ist hier neu?

- Wir änderten das Makefile.PL. In diesem Falle, spezifizierten wir eine zusätzliche Bibliothek, die in die geteilte Bibliothek von der Erweiterung gelinkt wurde, namentlich die Mathematik Bibliothek libm. Wir werden später sehen wie man XSUBs verfasst, die jede Routine in einer Bibliothek aufrufen können.
- Das Argument, welches an die Funktion übergeben wurde, wird nicht zurückgeliefert als Rückgabewert; vielmehr wird es der Variable, die übergeben wurde, zugewiesen. Du wirst es vielleicht erraten haben als du gesehen hast, das der Rückgabewert vom Typ 'void' war.

2.7 Ein-/Ausgabe Parameter

Du spezifizierst, welche Parameter an die XSUB übergeben werden auf der Zeile, die nach der Definition der Funktion's Rückgabewert und Name folgt. Jede Eingabe Parameter Zeile beginnt mit einem optionalen Leerzeichen und kann ein ebenso optionales terminierendes Semikolon enthalten.

Die Liste der Ausgabe Parameter befindet sich am Ende der Funktion, gerade vor der OUTPUT: Direktive. Die Verwendung von RETVAL teilt Perl mit, dass es erwünscht ist, dass jener Wert zurückgeliefert wird als Rückgabewert der XSUB Funktion. In Beispiel 3, war es erwünscht, dass der Rückgabewert in der originalen Variable platziert wurde; deshalb wurde sie in der OUTPUT: Sektion aufgeführt.

2.8 Das XSUBPP Programm

Das **xsubpp** Programm liest den XS Code in der .xs Datei und übersetzt es in C Code, ausgegeben in eine Datei deren Suffix .c ist. Der generierte C Code verwendet übermässig stark die C Funktionen innerhalb Perl.

2.9 Die TYPEMAP Datei

Das **xsubpp** Programm verwendet Regeln um von Perl's Datentypen (String, Array, etc.) zu C's Datentypen (int, char, etc.) zu konvertieren. Jene Regeln werden in der typemap Datei aufbewahrt ($PERLLIB/ExtUtils/typemap). Diese Datei wird in drei Teile unterteilt.

Die erste Sektion weist verschiedenen C Datentypen einem Namen zu, der mit den Perl Datentypen korrespondiert. Die zweite Sektion beinhaltet C Code, welcher **xsubpp** gebraucht, um Eingabe Parameter zu verarbeiten. Die dritte Sektion beinhaltet C Code, welcher **xsubpp** gebraucht, um Ausgabe Parameter zu verarbeiten.

Mögen wir einen Blick auf den Teil der .c Datei werfen, die für unsere Erweiterung kreiert wurde. Der Dateiname ist Mytest.c:

```
XS(XS_Mytest_round)
{
    dXSARGS;
    if (items != 1)
        croak("Usage: Mytest::round(arg)");
    {
        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 */
    }
    XSRETURN(1);
}
```

Man nehme die beiden Zeilen kommentiert mit 'XXXXX' zur Kenntnis. Wenn du die erste Sektion der typemap Datei prüfst, wirst du sehen, das doubles vom Typ T_DOUBLE sind. In der INPUT Sektion wird ein Argument, welches vom Typ T_DOUBLE ist, der Variable arg zugewiesen indem die Routine SvNV oder ähnlich aufgerufen, dann zu double gecastet und schlussendlich der Variable arg zugewiesen wird. Ähnliches geschieht in der OUTPUT Sektion; sobald arg seinen finalen Wert hat, wird es der sv_setnv Funktion übergeben, um es an die aufzurufende Subroutine zurückzugeben.

2.10 Warnungen betr. Ausgabe Argumenten

Generell gesprochen, ist es keine gute Idee Erweiterungen zu schreiben, welche die Eingabe Parameter ändern, wie in Beispiel 3. Stattdessen, sollte man wahrscheinlich multiple Werte in einem Array retournieren und den Caller jene verarbeiten lassen. Wie auch immer, um uns besser an existierende C Routinen anzupassen, welche oftmals ihre Eingabe Parameter ändern, wird jenes Verhalten toleriert.

2.11 Beispiel 4

In diesem Beispiel werden wir beginnen XSUBs zu schreiben, welche mit vordefinierten C Bibliotheken interagieren werden. Zu Beginn weg werden wir selber eine kleine Bibliothek schreiben, dann lassen wir h2xs unsere .pm und .xs Dateien erzeugen.

Erstelle ein neues Verzeichnis namens Mytest2 auf der selben Stufe wie das Verzeichnis Mytest. Im Mytest2 Verzeichnis erzeugst du ein Verzeichnis namens mylib und wechselst dann in jenes.

Nun kreieren wir einige Dateien, welche eine Test Bibliothek generieren werden. Jene werden eine C Quelldatei inkludieren und eine Headerdatei. Wir werden auch ein Makefile.PL in diesem Verzeichnis erzeugen. Anschliessend stellen wir sicher, dass wenn make in der Mytest2 Stufe ausgeführt wird automatisch die Makefile.PL Datei ausführt und das resultierende Makefile.

Erzeuge im mylib Verzeichnis eine Datei namens mylib.h, welche wie folgt aussieht:

```
#define TESTVAL 4

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

Erzeuge auch eine Datei namens mylib.c, die wie folgt aussieht:

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

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

Und schlussendlich, erzeuge eine Datei namens Makefile.PL, die wie folgt aussieht:

```
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)

    ';
}
```

Vergewissere dich, dass du Tabulatoren und keine Leerzeichen in den Zeilen mit $(AR) und $(RANLIB) beginnend verwendest. Make funktioniert nicht richtig, wenn du Leerzeichen verwenden solltest. Es wurde auch berichtet, dass das cr Argument für $(AR) auf Win32 Systemen unnötig ist.

Wir werden nun die Hauptdateien der obersten Stufe erstellen. Wechsele ins Verzeichnis über Mytest2 und führe folgenden Befehl aus:

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

Dies wird eine Warnung bezüglich dem überschreiben von Mytest2 ausgeben, aber dies ist akzeptabel. Unsere Dateien befinden sich in Mytest2/mylib und bleiben unverändert.

Das übliche Makefile.PL, welches h2xs generiert, weiss nichts von dem mylib Verzeichnis. Es ist vonnöten anzugeben das ein Unterverzeichnis existiert und dass wir darin eine Bibliothek erzeugen werden. Füge das Argument MYEXTLIB dem WriteMakefile Aufruf hinzu, sodass es folgendermassen aussieht:

```
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)',
);
```

Füge nun am Ende ein Subroutine hinzu (welche die vorherige überschreibt). Nehme zur Kenntnis, dass ein Tabulator für die Einrückung der Zeile mit cd beginnend vonnöten ist.

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

Ändere auch die MANIFEST Datei, sodass sie akkurat den Inhalt deiner Erweiterung widerspiegelt. Die einzelne Zeile, welche "mylib" enthält sollte durch folgende drei Zeilen ersetzt werden:

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

Um unseren Namensraum rein und unverschmutzt zu halten, editiere die .pm Datei und ändere die Variable `@EXPORT` zu `@EXPORT_OK`. Abschliessend editiere die .xs Datei, sodass die include Zeile folgendermassen ausschaut:

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

Füge auch folgende Funktionsdefinition der .xs Datei am Ende an:

```
double
foo(a,b,c)
        int             a
```

```
        long            b
        const char *    c
    OUTPUT:
        RETVAL
```

Nun müssen wir auch eine typemap Datei erzeugen, da das Standard Perl momentan den const char * Typ nicht unterstützt. Erstelle eine Datei namens typemap im Mytest2 Verzeichnis und füge folgendes ein:

```
const char *    T_PV
```

Führe nun das Makefile.PL im obersten Verzeichnis aus. Nehme zur Kenntnis, dass es auch ein Makefile im mylib Verzeichnis erstellt hat. Führe make aus und beobachte, dass es ins mylib Verzeichnis wechselt, um dort make auch auszuführen.

Editiere nun das test.pl Script und ändere den BEGIN Block, sodass er 1..4 ausgibt; desweiteren füge folgende Zeilen dem Ende des Scripts an:

```
print &Mytest2::foo(1, 2, "Hello, world!") == 7 ? "ok 2\n" : "not ok 2\n";
print &Mytest2::foo(1, 2, "0.0") == 7 ? "ok 3\n" : "not ok 3\n";
print abs(&Mytest2::foo(0, 0, "-3.4") - 0.6) <= 0.01 ? "ok 4\n" : "not ok 4\n";
```

Wenn man mit Fliesskommazahl Vergleichen zu tun hat, ist es am besten nicht auf Gleichheit zu prüfen, sondern viel eher ob die Differenz zwischen dem erwarteten und dem eigentlichen Resultat unter einer gewissen Schwelle liegt, die in diesem Falle 0.01 ist.

Führe `make test` aus und alles sollte in Ordnung sein.

2.12 Was geschah hier?

Nicht wie in vorherigen Beispielen haben wir h2xs an einer realen Include Datei getestet. Dies verursachte, dass einiges positives sowohl in der .pm wie auch .xs Datei gefunden werden kann.

- In der .xs Datei ist nun eine include Direktive mit dem absoluten Pfad zur mylib.h Header Datei. Wir haben jenen Pfad in einen relativen geändert, sodass wir des Erweiterungs Verzeichnis umbennen könnten, sofern wir dies möchten.
- Es wurde einiger neuer C Code der .xs Datei hinzugefügt. Der Zweck der `constant` Routine ist es die Werte welche in der Header Datei definiert sind dem Perl Script zugänglich zu machen (entweder durch aufrufen von `TESTCAL` oder `\&Mytest2::TESTVAL`). Es gibt auch XS Code um Aufrufe an die `constant` Routine zuzulassen.

- Die .pm Datei exportierte ursprünglich das Symbol `TESTVAL` im `@EXPORT` Array. Dies könnte Namenskollisionen hervorrufen. Eine gute Faustregel ist es, dass wenn define nur von C Routinen verwendet wird und nicht vom Benutzer selbst, es aus dem `@EXPORT` Array entfernt werden sollte. Nebst dem, wenn es dich nicht weiter stören sollte den vollständig qualifizierten Namen einer Variable zu gebrauchen, kannst du die meisten Elemente aus dem `@EXPORT` Array nach `@EXPORT_OK` zügeln.

- Hätte unsere Header Datei include Direktiven erhalten, wären sie nicht ordnungsgemäss durch h2xs verarbeitet worden. Es gibt keine wirklich gute Lösung dbzgl. momentan.

- Wir haben Perl auch von der Bibliothek, die wir im mylib Unterverzeichnis erstellt haben, mitgeteilt. Dies benötigte nur das hinzufügen der `MYEXTLIB` Variable an den WriteMakefile Aufruf und das ersetzen der Postamble Subroutine, sodass es ins Unterverzeichnis wechselt und make ausführt. Das Makefile.PL für die Bibliothek ist ein wenig mehr komplexer, aber nicht exzessiv. Wir haben soeben wieder die Postamble Subroutine durch unseren eigenen Code ersetzt. Jener Code spezifizierte ganz einfach, dass die Bibliothek, die erstellt werden sollte eine statische Archiv Bibliothek ist (im Gegensatz zur dynamisch-ladbaren Bibliothek) und fügte die Befehle, um sie zu compilieren, an.

2.13 Anatomie einer .xs Datei

Die .xs Datei in Beispiel 4 bestand aus einigen neuen Elementen. Um die Bedeutung jener Elemente zu verstehen, ist Aufmerksamkeit gegenüber folgender Zeile geboten:

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

Alles vor dieser Zeile ist C Code, der beschreibt, welche Header Dateien zu inkludieren sind, und einige Funktionen definiert. Keine Übersetzungen werden hiermit durchgeführt, nebst dem überspringen von eingebetteter POD Dokumentation wird alles so wie es ist in die generierte C Ausgabedatei eingefügt.

Alles nach dieser Zeile beschreibt XSUB Funktionen. Diese Beschreibungen werden übersetzt von **xsubpp** in C Code, welcher diese Funktionen - Perl Aufrufkonventionen gebrauchend - implementiert und jene für den Perl Interpreter sichtbar macht.

Schenke der Funktion `constant` deine Aufmerksamkeit. Jener Name erscheint zweimal in der generierten .xs Datei: einmal im ersten Teil, als statische C Funktion, dann ein anderes Mal im zweiten Teil, wenn eine XSUB Schnittstelle zur statischen C Funktion definiert wird.

Dies ist typisch für .xs Dateien: normalerweise stellt die .xs Datei eine Schnittstelle

zu einer existierenden C Funktion zur Verfügung. Dann wird jene C Funktion andersweitig definiert (entweder in einer externen Bibliothek oder im ersten Abschnitt einer .xs Datei) und eine Perl Schnittstelle zu jener Funktion wird im zweiten Teil der .xs Datei beschrieben. Die Situation wie sie in Beispiel 1, Beispiel 2 und Beispiel 3 vorkommt, wenn gänzlich alle Arbeit innerhalb der Perl Ebene getan wird, ist viel eher eine Ausnahme als die Regel.

2.14 XSUBs ausreizen

In Beispiel 4 enthält der zweite Teil der .xs Datei die folgende Beschreibung einer XSUB:

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

Nehme zur Kenntnis das im Gegensatz zu Beispiel 1, Beispiel 2 und Beispiel 3 die Beschreibung keinerlei aktuellen Code enthält, der beschreibt, was getan werden soll wenn die Perl Funktion foo() aufgerufen wird. Um Verständnis zu erlangen was hier vorgeht, füge folgende CODE Sektion der XSUB an:

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

Wie dem auch sei, diese beiden XSUBs stellen beinahe identisch generierten Code zur Verfügung: der **xsubpp** Compiler ist ausreichend intelligent, um die `CODE:` Sektion von den ersten beiden Zeilen der Beschreibung einer XSUB auszumachen. Wie siehts bzgl. `OUTPUT:` Sektion aus? Es ist identisch. Die `OUTPUT:` Sektion kann ebenfalls entfernt werden, sofern keine `CODE:` oder `PPCODE:` Sektion spezifiziert wird: **xsubpp** erkennt, dasses eine Funktionsaufruf Sektion generieren muss und wird die `OUTPUT:` Sektion ebenfalls automatisch erstellen. Folgendermassen kann die XSUB auf folgendes reduziert werden:

```
double
foo(a,b,c)
        int             a
```

```
        long            b
        const char *    c
```

Lässt sich selbiges mit einer XSUB

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

aus Beispiel 2 machen? Um jenes zu erreichen, müssen wir eine C Funktion `int is_even(int input)` definieren. Wie wir in der Anatomie einer .xs Datei sahen, ist die geeignete Stelle für jene Definition der erste Abschnitt einer .xs Datei. In der Tat eine C Funktion

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

ist wahrscheinlich zuviel des Guten. Etwas so simples wie ein `define` wird's auch tun:

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

Nachdem wir nun jenes als ersten Abschnitt in der .xs Datei haben, wird der Perl spezifische Teil einfach wie folgendermassen:

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

Diese Technik der Separierung der Schnittstelle vom eigentlichen Code, der die Arbeit tut, hat gewisse Nachteile: wenn man die Perl Schnittstelle ändern will, muss man sie an zwei Stellen im Code ändern. Wie dem auch sei, es entfernt sehr viel unleserlichen Code und macht den eigentlichen Code unabhängig von den Aufrufkonventionen in Perl. (In der Tat, es ist nichts Perl-spezifisches in der obigen Beschreibung; eine verschiedene **xsubpp** Version hätte vielleicht dies u.U. zu TCL oder Python konvertiert.)

2.15 Weitere XSUB Argumente

Mit der Vervollständigung des Beispiel 4, ist es nun ein einfaches Bibliotheken zu simulieren, die im reellen Leben vokommen, deren Schnittstellen wahrscheinlich nicht die fehlerfreiesten sind. Wir werden nun fortfahren mit einer Erläuterung der Argumente, die dem **xsubpp** Compiler übergeben werden.

Wenn du Argumente an Routinen in einer .xs Datei spezifizierst, übergibst du in Tat und Wahrheit drei Stücke an Information für jedes aufgelistete Argument. Das erste Stück ist das Argument relativ zu den andern (erstes, zweites, etc.). Das zweite ist der Typ des Argument und besteht aus der Typ Deklaration des Arguments (dh. int, char*, etc.). Das dritte ist die Aufrufkonvention für das Argument das an an die Bibliotheksfunktion übergeben wird.

Während Perl Argumente an Funktionen via Referenz übergibt, übergibt sie C als Wert; um eine C Funktion zu implementieren, die die Daten eines Arguments ändert, würde das aktuelle Argument dieser C Funktion ein Zeiger auf Daten sein. Deswegen werden zwei C Funktionen mit den Deklaration

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

gänzlich verschiedene Semantiken haben: das erste wird wohl ein Array von Zeichen referenziert durch s inspektieren, zweiteres könnte unmittelbar `cp` derefenzieren und nur `*cp` manipulieren (den Rückgabewert als Erfolgsindikator gebrauchend). Von Perl aus würden jene Funktion komplett anders verwendet werden.

Diese Information an **xsubpp** wird überbracht indem `*` durch `\&` vor dem Argument ersetzt wird. `\&` bedeutet, dass das Argument an eine Bibliotheksfunktion via Addresse übergeben werden soll. Die beiden obigen Funktionen werden XSUB-ified als

```
int
string_length(s)
        char * s

int
upper_case_char(cp)
        char &cp
```

Beispielsweise, bedenke:

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

Das erste Perl Argument an diese Funktion würde als char behandelt werden, an die Variable a zugewiesen werden und die Adresse würde auch an die Funktion übermittelt werden. Das zweite Perl Argument würde als String Referenz behandelt werden und an die Variable b zugewiesen. Der Wert von b würde an die Funktion foo weitergereicht werden. Der eigentliche Aufruf der Funktion foo, die **xsubpp** generiert, würde folgendermassen ausschauen:

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

xsubpp würde folgende Funktionsargumente Liste identisch verarbeiten:

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

Wie dem auch sei, um das Verständnis zu erleichtern, wird es empfohlen, dass man ein '&' nebst dem Variabelnamen platziert, aber separiert vom Variabelntyp und dass man ein '*' nebst dem Variabelntyp platziert, aber separiert vom Variabelnamen (wie im obigen Aufruf). Indem man dies tut, ist es einfach zu verstehen, was exakt an die C Funktion übergeben wird - was auch immer in der letzten Spalte ist.

Du solltest grosse Anstrengungen unternehmen, um zu versuchen der Funktion den Typ der Variable zu übergeben, der erwartet wird, wenn möglich. Es wird dich vor grösseren Schwierigkeiten bewahren.

2.16 Der Argument Buffer

Wenn wir irgendein Stück generierten C Code betrachten ausser Beispiel 1, wirst du eine Anzahl an Referenzen an ST(n) zur Kenntnis nehmen, wo n üblicherweise 0 ist. ST ist eigentlich ein Makro, welches zum n'ten Argument auf dem Argument Buffer zeigt. ST(0) ist foldendermassen das erste Argument auf dem Buffer und deswegen das erste Argument, welches an die XSUB übergeben wird, ST(1) das zweite Argument, usw.

Wenn du die Argumente an die XSUB in der .xs Datei auflistest, teilt dies **xsubpp** mit welches Argument mit welchem auf dem Argument Buffer korrespondiert (dh. das erste aufgelistet ist das erste Argument usw.). Du wirst Schwierigkeiten verursachen, wenn du sie nicht in der selben Reihenfolge auflisten in der die Funktion sie erwartet.

Die eigentlichen Werte auf dem Argument Buffer sind Zeiger zu den übergebenen Werten. Wenn ein Argument als OUTPUT Wert aufgelistet wird, wird sein korrespondierender Wert auf dem Buffer (dh. ST(0) wenn es das erste Argument war) geändert. Du kannst dies verfizieren indem du den generierten C Code für Beispiel 3 begutachtest. Der Code für die round() XSUB Routine beinhaltet Zeilen, die wie folgt aussehen:

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

Die arg Variable wird initial auf den Wert von ST(0) gesetzt und am Ende der Routine wird sie zurück in ST(0) geschrieben.

XSUBs ist es auch gestattet Listen zu retournieren, nicht nur Skalare. Dies muss realisiert werden indem Buffer Werte ST(0), ST(1), etc. in einer subtilen Art & Weise manipuliert werden.

XSUBs ist es überdies gestattet eine automatische Konversion von Perl Funktionsargumenten zu C Funktionsargumenten zu vermeiden. Gewisse Leute ziehen manuelle Konversion vor indem sie `ST(i)` überprüfen sogar in Fällen, wenn automatische Konversion ausreichend wäre, argumentierend das es die Logik eines XSUB Aufruf vereinfacht. Man vergleiche es mit 'XSUBs ausreizen' für einen ähnlichen Gewinn anhand einer kompletten Separierung der Perl Schnittstelle und des eigentlichen Codes einer XSUB.

Wenn auch Experten über jene Idiome argumentieren, ein Anfänger betreffend den Perl Innereien wird eine Lösung bevorzugen, welche so wenig wie möglich Perl-Innereien spezifisch ist, die automatische Konversion und Generation von Aufrufen bevorzugend, wie in 'XSUBs ausreizen'. Diese Herangehensweise hat den zusätzlichen Vorteil, dass es den XSUB Verfasser von zukünftigen Änderungen an der Perl API bewahrt.

2.17 Erweitern der Erweiterung

Manchmal beabsichtigt man zusätzliche Methoden oder Subroutinen zur Verfügung zu stellen, um das Verständnis um die Funktionsweise der Schnittstelle zwischen Perl und Ihrer Erweiterung zu vereinfachen. Jene Funktionen sollten in der .pm Datei enthalten sein. Ob sie automatisch geladen werden wenn die Erweiterung geladen wird oder nur wenn sie explizit aufgerufen werden, ist abhängig von der Platzierung der Subroutine Definition in der .pm Datei. Du solltest auch die AutoLoader Dokumentation konsultieren für einen alternativen Weg um Subroutinen zu speichern und laden.

2.18 Dokumentation der Erweiterung

Es gibt keine Rechtfertigung für das auslassen der obligaten Dokumentation. Dokumentation gehört üblicherweise in die .pm Datei. Der Inhalt jener Datei wird an pod2man übergeben, die eingebettete Dokumentation wird ins manpage Format konvertiert und im blib Verzeichnis platziert. Es wird in Perl's manpage Verzeichnis kopiert sobald die Erweiterung installiert wird.

Man möge Dokumentation und Perl Code innerhalb einer .pm Datei durchmischen. In der Tat, wenn man Methode Autoladen einsetzen will, muss man dies tun, wie der Kommentar innerhalb der .pm Datei erläutert.

2.19 Installation der Erweiterung

Einst wenn die Erweiterung komplettiert wurde und alle Tests erfolgreich sind, ist die Installation ein einfaches: man führe ganz einfach make install aus. Du wirst entweder Schreibrechte für die Verzeichnisse benötigen wo Perl installiert wird oder deinen Systemadministrator fragen müssen, ob er make für dich ausführt.

Alternativ kann man das exakte Verzeichnis spezifizieren, wo die Dateien der Erweiterungen platziert werden, indem man ein PREFIX=/destination/ directory nach make install anhängt. Dies kann sich als sehr nützlich erweisen wenn du eine Erweiterung compilierst, welche eventuell für mehrere Systeme veröffentlicht wird. Du kannst dann einfach die Dateien im Zielverzeichnis archivieren und sie auf deine Zielsysteme kopieren.

2.20 Beispiel 5

In diesem Beispiel werden wir uns noch mehr mit dem Argument Buffer auseinandersetzen. All vorherigen Beispiele haben jeweils nur einen einzelnen Wert retourniert. Wir kreieren nun eine Erweiterung, welche ein Array retourniert.

Diese Erweiterung ist sehr UNIX-fokussiert (struct statfs und der statfs Systemaufruf). Wenn du nicht ein Unix System betreibst, kannst du statfs mit einer Funktion ersetzen, die mehrere Werte retourniert, du kannst Werte einfügen die an den Aufrufer zurückgegeben werden sollen, oder du kannst schlicht und einfach dieses Beispiel überspringen. Sofern du die XSUB änderst, stelle sicher, dass die Tests angepasst werden, sodass sie den Änderungen entsprechen.

Kehre ins Mytest Verzeichnis zurück und füge folgenden Code dem Ende von Mytest.xs an:

```
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)));
        XPUSHs(sv_2mortal(newSVnv(buf.f_fsid[0])));
        XPUSHs(sv_2mortal(newSVnv(buf.f_fsid[1])));
    } else {
        XPUSHs(sv_2mortal(newSVnv(errno)));
    }
```

Du benötigst ausserdem den folgenden Code am Beginn der .xs Datei, soeben nach dem include der 'XSUB.h' Datei:

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

Füge ausserdem folgendes Code Segment der Datei test.pl an und inkrementiere den 1..9 String im BEGIN Block zu 1..11:

```
@a = &Mytest::statfs("/blech");
print ((scalar(@a) == 1 && $a[0] == 2) ? "ok 10\n" : "not ok 10\n");
@a = &Mytest::statfs("/");
print scalar(@a) == 9 ? "ok 11\n" : "not ok 11\n";
```

2.21 Neues in diesem Beispiel

Dieses Beispiel erweckte einige neue Konzepte zum Leben. Wir werden sie sukzessiv betrachten.

- Die INIT: Direktive enthält Code welcher unmittelbar platziert wird nachdem der Argument Buffer dekodiert wird. C erlaubt keine Variabelndeklarationen an eigenmächtigen Positionen innerhalb einer Funktion, deshalb ist dies üblicherweise der beste Weg um lokale Variabeln, die von der XSUB gebraucht werden, zu deklarieren. (Alternativ, könnte man die `PPCDOE:` Sektion gänzlich in Klammern fassen und jene Deklarationen oben platzieren.)

- Diese Routine retourniert auch eine differierende Anzahl an Argumenten basierend auf dem Erfolg oder Miserfolg des statfs Aufrufs. Sofern ein Fehler vorliegen sollte, wird die Fehlernummer als ein Array mit einem Element retourniert. Sofern der Aufruf erfolgreich war, wird ein Array mit 9 Elementen retourniert. Da nur ein Argument an jene Funktion übergeben wird, benötigen wir Platz im Buffer, der die 9 Werte, die retourniert werden, fassen soll.

  Wir tun dies indem wir die PPCODE: Direktive gebrauchen, statt der CODE: Direktive. Dies teil **xsubpp** mit, dass wir die Rückgabewerte, die wir auf den Argument Buffer tun, selber verwalten werden.

- Sofern wir beabsichtigen Werte, welche retourniert werdens sollen, auf den Buffer zu packen, gebrauchen wir eine Serie von Makros, welche mit XPUSH beginnen. Es existieren fünf verschiedene Versionen, um integers, unsigned integers, doubles, Strings und Perl Skalare auf dem Buffer zu platzieren. In unserem Beispiel, platzierten wir ein Perl Skalar auf dem Buffer. (Dies ist in der Tat das einzige Makro, welches verwendet werden kann um mehrfache Werte zu retournieren.)

  Die XPUSH* Makros werden automatisch den Rückgabe Buffer erweitern, um einen Overrun zu vermeiden. Sie platzieren Werte in der Reihenfolge auf dem Buffer wie du es wüenschst, sodass es das aufzurufende Programm sieht.

- Die Werte, die auf dem Rückgabe Buffer platziert werden, sind eigentliche 'sterbliche' SV's. Sie werden sterblich gemacht, sodass einst die Werte vom aufzurufenden Programm kopiert worden sind, die SV's, welche die Rückgabewerte enthielten, dealloziert werden können. Wären sie nicht sterblich, würden sie nachdem die XSUB Routine zurückgekehrt ist weiter existieren, wären aber nicht mehr zugänglich. Das ist ein Speicherleck.

- Wären wir in Geschwindigkeit, nicht in Code Kompaktheit interessiert, würden wir im Erfolgsfalle nicht `XPUSHs` Makros gebrauchen, sondern `PUSHs` Makros; desweiteren würden wir vorgängig den Buffer erweitern bevor wir Rückgabewerte darauf platzieren:

  ```
  EXTEND(SP, 9);
  ```

- Der Kompromiss ist das man die Anzahl Rückgabewerte im voraus kalkulieren muss (obschon übermassig den Buffer erweiternd wird niemanden ausser der Speicherauslastung selbst schaden).

  Ähnlicherweise, im Miserfolgsfalle könnten wir `PUSHs` verwenden ohne den Buffer zu erweitern: die Perl Funktionsreferenz zeigt auf eine XSUB auf dem Buffer, folgendermassen ist der Buffer immer gross genug um mindest ein Rückgabewert zu fassen.

2.22 Beispiel 6

In diesem Beispiel werden wir eine Referenz zu einem Array als Eingabe Parameter akzeptieren und retournieren eine Referenz zu einem Array mit Hashes. Dies wird die Manipulation komplexer Perl Datentypen von einer XSUB ausgehend demonstrieren.

Diese Erweiterung ist irgendwie ausgedacht. Sie basiert auf dem Code des vorherigen Beispiels. Es ruft die statfs Funktion mehrere Male auf, akzeptiert eine Referenz zu einem Array aus Dateinamen bestehend als Eingabe und retourniert

eine Referenz zu einem Array mit Hashes, welche die Daten für jedes Dateisystem enthält.

Kehre zum Mytest Verzeichnis zurück und füge folgenden Code dem Ende von Mytest.xs an:

```
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
```

Füge auch den folgenden Code dem test.pl hinzu und inkrementiere den 1..11 String im BEGIN Block zu 1..13.

```
$results = Mytest::multi_statfs([ '/', '/blech' ]);
print ((ref $results->[0]) ? "ok 12\n" : "not ok 12\n");
print ((! ref $results->[1]) ? "ok 13\n" : "not ok 13\n");
```

2.23 Neues in diesem Beispiel

Einige neue Konzepte werden hier eingeführt, wie nachfolgend beschrieben:

- Diese Funktion gebraucht keine typemap. Stattdessen deklarieren wir sie, sodass sie ein SV* (Skalar) Parameter akzeptiert und ein SV* Wert retourniert; desweiteren, tragen wir Sorge, dass jene Skalare innerhalb des Codes ins Leben gerufen werden. Da wir nur ein Wert retournieren, brauchen wir keine `PPCODE:` Direktive - stattdessen gebrauchen wir die `CODE:` und `OUTPUT:` Direktiven.

- Wenn wir mit Referenzen zu tun haben, ist es wichtig jenen mit Vorsicht zu begegnen. Der `INIT:` Block prüft initial ob `SvROK` wahr retourniert, was indiziert, dass paths eine gültige Referenz ist. Es prüft dann ob das Objekt, welches durch paths referenziert wird, ein Array ist, indem es `SvRV` gebraucht um paths zu dereferenzieren und `SvTYPE` um den Typ zu bestimmen. Wenn es als Test hinzugefügt wird, prüft es ob das Array durch paths referenziert ungleich leer ist, indem es die `av_len` Funktion gebraucht (welche -1 retourniert, wenn der Array leer ist). Das XSRETURN_UNDEF Makro wird verwendet um die XSUB zu verlassen und den undefinierten Wert zu retournieren, sofern all jene drei Bedingungen nicht zutreffen.

- Wir manipulieren mehrere Arrays in der XSUB. Nehme zur Kenntnis, dass ein Array intern durch ein AV* Zeiger repräsentiert wird. Die Funktionen und Makros, um Arrays zu manipulieren ähneln den folgenden Perl Funktionen: `av_len` retourniert den höchsten Index in einem AV*; `av_fetch` liefert einen einzelnen Wert aus dem Array zurück, der mit dem überlieferten Index korrespondiert; `av_push` fügt einen skalaren Wert am Ende des Arrays hinzu, automatisch das Array erweiternd, sofern es vonnöten ist.

  Wir lesen ausdrücklich Pfadnamen einzeln aus dem Eingabe Array und speichern die Resultate in einem Ausgabe Array in derselben Reihenfolge. Sollte statfs nicht erfolgreich sein, wird anstatt des Wertes die zugehörige Fehlernummer im Array gespeichert. Sollte statfs erfolgreich sein, wird der

Wert der im Array gespeichert wird eine Referenz auf einen Hash sein, der einige Informationen bzgl. der statfs Struktur beherbergt.

Den Rückgabe Buffer betreffend wäre es möglich (und ein kleiner Geschwindigkeits Gewinn), den Rückgabe Buffer proaktiv zu erweitern, bevor Werte in ihm gespeichert werden, da wir wissen wieviele Elemente wir retournieren werden:

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

- Wir führen nur eine Hash Operation in dieser Funktion durch, dh. wir speichern einen neuen Skalar unter einem Schlüssel `hv_store` gebrauchend. Ein Hash wird representiert durch ein HV* Zeiger. Wie für Arrays, widerspiegeln die Funktionen um Hashes innerhalb einer XSUB zu manipulieren, die Funktionalität verfügbar aus Perl.
- Um einen Zeiger zu erstellen, gebrauchen wir die `newRV` Funktion. Nehme zur Kenntnis, dass du ein AV* oder HV* zum Typ SV* (und vielen anderen) casten kannst. Dies ermöglicht Ihnen Zeiger auf Arrays, Hashes und Strings mit derselben Funktion zu kreieren. Hingegen retourniert die `SvRV` Funktion immer ein SV*, welches eventuell in den richtigen Typ gecastet werden muss, sofern es etwas anderes als ein String sein sollte (siehe `SvTYPE`).
- An dieser Stelle verrichtet xsubpp nur minimale Arbeit - die Differenzen zwischen Mytest.xs und Mytest.c sind minim.

2.24 Beispiel 7

Du könntest denken, Dateien an XS zu überreichen wäre schwierig, mit all jenen Typeglobs und Sachen. Nun denn, ist es nicht.

Nehme an, dass wir für einen merkwürdigen Grund einen Wrapper um die standardisierte C Bibliotheksfunktion `fputs()` benötigen. Nachfolgend alles was wir brauchen:

```
#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
```

Die wahre Arbeit wird durch das Standard typemap verrichtet.

Aber Du kommst nicht in den Genuss der Feinarbeit, die durch die perlio Schnittstellen getan wird. Hier wird die stdio Funktion `fputs()` aufgerufen, welche nichts davon weiss.

Das Standard typemap offeriert drei Varianten der PerlIO: `InputStream` (T_IN), `InOutStream` (T_INOUT) und `OutputStream` (T_OUT). Eine schlichte PerlIO* wird als T_INOUT bedacht. Wenn es dir in deinem Code drauf ankommt (siehe unten für Gründe), define'n oder typedef'e einer der spezifischen Namen und verwende jenen als Argument oder Resultatstyp in ihrer XS Datei.

Das Standard typemap enhält kein PerlIO* vor Perl 5.7, aber es gibt drei Stream Varianten. Ein PerlIO* direkt verwendet ist nicht abwärtskompatibel ausser man tut die eigene typemap zur Verfügung stellen.

Für Streams die von Perl aus kommen ist der Hauptunterschied, dass `OutputStream` den Ausgabe PerlIO* erhält - was u.U. einen Unterschied ausmacht auf einem Socket.

Für Streams, welche an Perl übergeben werden wird ein neues Filehandle kreiert (dh. ein Zeiger zu einem neuen Glob) und mit dem zur Verfügung gestellten PerlIO * assoziert. Sollte der lese/schreib Status der PerlIO * nicht korrekt sein, dann könntest du Fehler oder Warnungen erhalten, sofern das Filehandle gebraucht wird. Dh. wenn du das PerlIO * als "w" öffnen sollte es `OutputStream` sein, wenn als "r" geöffnet `InputStream`.

Nun nehmen wir an du willst perlio Schnittstellen in deinem XS gebrauchen. Wir verwenden die perlio `PerlIO_puts()` Funktion in diesem Beispiel.

In dem C Abschnitt der XS Datei (oben an der ersten MODULE Zeile) hast du folgendes:

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

Und im XS Code:

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

Wir müssen wir die `CODE` Sektion gebrauchen, da `PerlIO_puts()` dieselben Argumente umgekehrt verglichen mit `fputs()` enthält, und wir die Argumente als selbiges behalten wollten.

Um jenes weiter zu erforschen, müssen wir die stdio `fputs()` Funktion auf einem PerlIO * gebrauchen. D.h. wir müssen von dem perlio System ein stdio `FILE *` anfordern:

```
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
```

Bemerkung: `PerlIO_findFILE()` wird die Schnittstellen absuchen für eine stdio Schnittstelle. Sollte keine gefunden werden, wird es `PerlIO_exportFILE()` aufrufen, um ein neues stdio `FILE` zu generieren. Rufe nur `PerlIO_exportFILE()` auf wenn du ein neues `FILE` benötigst. Es wird nach jedem Aufruf eines generieren und der stdio Schnittstelle übergeben. Deswegen rufe es nicht mehrmals für die gleiche Datei auf. `PerlIO()_findFile` wird die stdio Schnittstelle erhalten wenn es durch `PerlIO_exportFILE` generiert wurde.

Dies trifft nur auf das perlio System zu. Für Versionen vor 5.7 ist `PerlIO_exportFILE()` äquivalent zu `PerlIO_findFILE()`.

2.25 Problembehebung

Wie am Anfang dieses Dokument erwähnt, solltest du Probleme mit den Erweiterungsbeispielen haben, siehe ob nachfolgende Ratschläge helfen.

- In Versionen 5.002 vor der Gamma Version, wird das Test Script in Beispiel 1 nicht ordnungsgemäss funktionieren. Du musst die üse libSZeile folgendermassen ändern:

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

- In Versionen 5.002 vor der Version 5.002b1h, wurde die test.pl Datei nicht automatisch durch h2xs erzeugt. Dies bedeutet, dass du nicht einfach make

test ausführen kannst, um das Test Script auszuführen. Du musst die folgende Zeile vor der "use extension" Anweisung platzieren:

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

- In Versionen 5.000 und 5.001, wirst du statt der obig ersichtlichen Zeile, folgende Zeile gebrauchen müssen:

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

- Dieses Dokument nimmt an das die ausführbare Datei namens "perl" Perl Version 5 ist. Einige Systeme haben Perl Version 5 eventuell als "perl5" installiert.

3 Siehe auch

Für weiterführende Informationen, ziehe perlguts, perlapi, perlxs, perlmod und perlpod zu Rate.

4 Autor

Jeff Okamato *okamoto@corp.hp.com*

Begutachtet und assistiert durch Dean Roehrich, Ilya Zakharevich, Andreas Koenig und Tim Bunce.

PerlIO Sachen hinzugefügt durch Lupe Christoph mit einiger Verbesserung durch Nick Ing-Simmons.

Übersetzt durch Steven Schubiger.

5 Letzte Änderung

2002/05/08