perlxstut

Table des matières

1 NAME/NOM

perlXStut - Guide d'apprentissage des XSUB

2 DESCRIPTION

Ce guide fournit au lecteur les étapes à suivre pour réaliser une extension de Perl. On suppose qu'il a accès à *perlguts* et à *perlxs*.

Le guide commence par des exemples très simples, avant d'aborder des points plus complexes, chaque nouvel exemple introduisant des fonctionnalités supplémentaires. Certains concepts ne seront pas expliqués complètement du premier coup, afin de faciliter au lecteur l'apprentissage progressif de la construction d'extensions.

2.1 VERSION DE CE DOCUMENT

Nous nous efforçons de maintenir ce guide à jour par rapport aux dernières versions de développement de Perl. Cela signifie qu'il est parfois en avance sur la dernière version stable de Perl, et qu'il se peut que des fonctionnalités décrites ici soient absentes dans les versions plus anciennes. Cette section conserve la trace des fonctionnalités ajoutées à Perl 5.

– Dans les versions de Perl 5.002 antérieures à la version gamma, le script de test de l'exemple 1 ne fonctionnera pas correctement. Vous devrez modifier la ligne "use lib" comme suit :

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

– Dans les versions de Perl 5.002 antérieures à la version beta 3, la ligne du fichier .xs contenant "PROTYPES: DISABLE" entraînera une erreur de compilation. Supprimez simplement cette ligne.

– Dans les versions de Perl 5.002 antérieures à la version 5.002b1h, le fichier test.pl n'était pas créé automatiquement par h2xs. Cela signifie que vous ne pouvez pas exécuter "make test" pour lancer le script de test. Vous devrez rajouter la ligne suivante avant l'instruction "use extension" :

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

– Dans les versions 5.000 et 5.001, vous ne devez pas utiliser la ligne qui précède, mais plutôt celle-ci :

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

– On suppose dans ce document que l'exécutable dont le nom est "perl" correspond à Perl, version 5. Dans certains systèmes, Perl 5 peut avoir été installé sous le nom "perl5".

2.2 DYNAMIQUE / STATIQUE

On croit souvent que, si un système ne supporte pas les librairies dynamiques, il est impossible d'y construire des XSUB. C'est une erreur. Vous *pouvez* les construire, mais, à l'édition de liens, vous devez assembler les sous-routines de la XSUB au reste de Perl à l'intérieur d'un nouvel exécutable. La situation est la même que dans Perl 4.

Ce guide peut quand même être utilisé sur un tel système. Le système de compilation de la XSUB détectera le système et construira, si cela est possible, une librairie dynamique ; sinon, il construira une librairie statique accompagnée, de manière optionnelle, d'un nouvel exécutable contenant cette librairie liée de manière statique.

Supposons que vous souhaitiez construire un exécutable lié statiquement, sur un système supportant les librairies dynamiques. Dans ce cas, chaque fois que la commande "make" sans argument est exécutée dans les exemples qui suivent, vous devrez utiliser la commande "make perl" à la place.

Si vous avez choisi de générer un tel exécutable lié statiquement, vous devrez aussi remplacer "make test" par "make test_static". Sur les systèmes qui ne peuvent pas du tout construire de librairie dynamique, "make test" est suffisant.

2.3 EXAMPLE 1

Notre première extension sera très simple. Lorsque nous appellerons la routine définie dans l'extension, elle affichera un message donné et se terminera.

Exécutez `h2xs -A -n Montest`. Cela crée un répertoire nommé Montest, éventuellement sous ext/ si ce répertoire existe dans le répertoire courant. Plusieurs fichiers seront créés sous Montest/, en particulier MANIFEST, Makefile.PL, Montest.pm, Montest.xs, test.pl et Changes.

Le fichier MANIFEST contient les noms de tous les fichiers créés.

Le fichier Makefile.PL devrait ressembler à ceci :

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

Le fichier Montest.pm devrait commencer à peu près comme suit :

```
package Montest;
```

```
require Exporter;
require DynaLoader;

@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.
@EXPORT = qw(

);
$VERSION = '0.01';

bootstrap Montest $VERSION;

# Preloaded methods go here.

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

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

Et voici à quoi devrait ressembler le fichier Montest.xs :

```
#ifdef __cplusplus
extern "C" {
#endif
#include "EXTERN.h"
#include "perl.h"
#include "XSUB.h"
#ifdef __cplusplus
}
#endif

PROTOTYPES: DISABLE

MODULE = Montest                PACKAGE = Montest
```

Modifions le fichier .xs en ajoutant ceci à la fin :

```
void
bonjour()
        CODE:
        printf("Bonjour !\n");
```

A présent, lançons "perl Makefile.PL", ce qui va créer un vrai Makefile, nécessaire pour make. Les résultats suivants seront affichés :

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

En lançant maintenant make, nous obtenons à peu près la sortie suivants (certaines lignes sont tronquées pour plus de clarté) :

```
% make
umask 0 && cp Montest.pm ./blib/Montest.pm
perl xsubpp -typemap typemap Montest.xs >Montest.tc && mv Montest.tc Montest.c
cc -c Montest.c
Running Mkbootstrap for Montest ()
chmod 644 Montest.bs
LD_RUN_PATH="" ld -o ./blib/PA-RISC1.1/auto/Montest/Montest.sl -b Montest.o
chmod 755 ./blib/PA-RISC1.1/auto/Montest/Montest.sl
cp Montest.bs ./blib/PA-RISC1.1/auto/Montest/Montest.bs
chmod 644 ./blib/PA-RISC1.1/auto/Montest/Montest.bs
```

A présent, bien qu'il y ait déjà un patron test.pl prêt à l'emploi, nous allons, pour cet exemple seulement, réaliser un script de test spécial. Créez un fichier avec le nom bonjour, qui contiendra ce qui suit :

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

use ExtUtils::testlib;

use Montest;

Montest::bonjour();
```

A ce point, en lançant le script, vous devriez obtenir la sortie suivante :

```
% perl bonjour
Bonjour !
%
```

2.4 EXAMPLE 2

Nous allons maintenant ajouter dans notre extension une sous-routine qui prendra un argument et renverra 1 si l'argument est pair, 0 s'il est impair.

Ajoutez les lignes suivantes à la fin de Montest.xs :

```
int
est_pair(entree)
        int     entree
        CODE:
        RETVAL = (entree % 2 == 0);
        OUTPUT:
        RETVAL
```

Les espaces blancs au début de la ligne "int entree" ne sont pas obligatoires, mais ils améliorent la lisibilité. Le point-virgule à la fin de la même ligne est aussi facultatif.

"int" et "entree" peuvent être séparés par des blancs. On pourrait aussi se passer d'indenter les quatre lignes à partir de celle qui commence par "CODE:". Mais il est recommandé, pour des raisons de lisibilité, d'utiliser une indentation de 8 espaces (ou une tabulation normale).

A présent, relancez make pour reconstruire la librairie partagée.

Puis suivez à nouveau les étapes ci-dessus pour générer un Makefile à partir du fichier Makefile.PL et lancer make.

Pour vérifier que l'extension fonctionne, nous devons regarder le fichier test.pl. Ce fichier est organisé de manière à simuler le système de tests de Perl lui-même. Dans ce script, vous lancez une série de tests qui vérifient le comportement de l'extension, en affichant "ok" lorsque le test réussit, "not ok" dans le cas contraire. Modifiez l'instruction print dans le bloc BEGIN pour afficher "1..4", et ajoutez le code suivant à la fin du fichier :

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

Nous allons appeler le script de test avec la commande "make test". Vous devriez obtenir une sortie comme la suivante :

```
% make test
PERL_DL_NONLAZY=1 /opt/perl5.002b2/bin/perl (nombreux arguments -I) test.pl
1..4
ok 1
ok 2
ok 3
ok 4
%
```

2.5 QUE S'EST-IL DONC PASSE ?

Le programme h2xs est le point de départ de la création d'extensions. Dans les exemples à venir, nous verrons comment utiliser h2xs pour lire des fichiers d'en-tête et générer des patrons pour se connecter à des routines en C.

h2xs crée plusieurs fichiers dans le répertoire de l'extension. Le fichier Makefile.PL est un script Perl qui va générer un vrai Makefile pour construire l'extension. Nous verrons ce point plus en détail dans un moment.

Les fichiers <extension>.pm et <extension>.xs contiennent le coeur de l'extension. Dans le fichier .xs se trouvent les routines en C qui constitueront l'extension. Le fichier .pm contient des routines qui indiqueront à Perl comment charger l'extension.

En générant puis en invoquant le Makefile, on crée un répertoire blib (c'est-à-dire "binary library" ou librairie binaire) dans le répertoire courant. Ce répertoire contiendra la librairie partagée que nous construirons. Après l'avoir testée, nous pourrons installer celle-ci à son emplacement final.

L'utilisation de "make test" pour lancer le programme de test fait quelque chose de très important : cette commande appelle Perl avec tous les arguments `-I`, ce qui lui permet de trouver les divers fichiers constituant l'extension.

Il est *primordial* d'utiliser "make test" tant que vous êtes encore en train de tester les extensions. Si vous essayez de lancer le script de test directement, vous aurez une erreur fatale.

Une autre raison de l'importance de passer par "make test" pour lancer votre script de test est que, lorsque vous testez la mise à jour d'une version existante, "make test" garantit que vous utilisez votre nouvelle extension et non la version existante.

Lorsque Perl voit un `use extension;`, il cherche un fichier qui ait le même nom que l'extension faisant l'objet du `use`, avec un suffixe .pm. S'il ne peut pas trouver ce fichierx, Perl s'arrête avec une erreur fatale. Le chemin de recherche par défaut est contenu dans le tableau @INC.

Dans notre cas, Montest.pm indique à Perl qu'il aura besoin des extensions Exporter et Dynamic Loader. Puis il remplit les tableaux @ISA et @EXPORT, ainsi que le scalaire $VERSION ; finalement, il demande à Perl d'initialiser l'extension. Perl appellera la routine de chargement dynamique (si elle existe) et chargera la librairie dynamique.

Les deux tableaux qui sont positionnés dans le fichier .pm ont une importance particulière. @ISA contient une liste de paquetages où doivent être recherchées les méthodes (ou les sous-routines) qui n'existent pas dans le paquetage courant. Le tableau @EXPORT indique à Perl quelles routines, parmi celles de l'extension, doivent être placées dans l'espace de nommage du paquetage appelant.

Il faut vraiment choisir avec soin ce qu'on exporte. N'exportez JAMAIS des noms de méthodes, et n'exportez PAS autre chose *par défaut* sans une bonne raison.

En règle générale, si le module est orienté objet, n'exportez rien du tout. S'il s'agit seulement d'un ensemble de fonctions, vous pouvez alors exporter n'importe quelle fonction dans un autre tableau, @EXPORT_OK.

Consultez *perlmod* pour plus d'informations.

La variable $VERSION a pour rôle de garantir que le fichier .pm et la librairie partagée sont "en phase" l'un avec l'autre. Chaque fois que vous modifiez le fichier .pm ou .xs, vous devriez incrémenter la valeur de cette variable.

2.6 ECRIRE DE BONS SCRIPTS DE TEST

On ne dira jamais trop combien il est important de rédiger de bons programmes de test. Vous devriez suivre de près le style "ok/not ok" utilisé par Perl lui-même, afin que chaque test puisse être interprété très facilement et sans ambiguïté. Lorsque vous trouvez un bug et que vous le corrigez, n'oubliez pas de rajouter un test unitaire à son sujet.

En lançant "make test", vous vous assurez que le script test.pl fonctionne et qu'il utilise votre extension dans la bonne version. Si vous avez un grand nombre de tests unitaires, peut-être voudrez-vous imiter la structure des fichiers de tests de Perl. Créez un répertoire nommé "t", et donnez à tous vos scripts de test la terminaison ".t". Le Makefile lancera alors tous ces tests de la manière adéquate.

2.7 EXAMPLE 3

Notre troisième extension prendra un argument en entrée, arrondira sa valeur et la remettra dans l'*argument* lui-même.

Rajoutez le code qui suit à la fin de Montest.xs :

```
void
arrondir(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
```

Modifiez le fichier Makefile.PL de manière à ce que la ligne "LIBS" ressemble à ce qui suit :

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

Générez le Makefile, et lancez make. Dans test.pl, modifiez le bloc BEGIN afin d'afficher "1..9", et ajoutez ceci :

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

"make test" devrait à présent dire que les neuf tests sont réussis.

Vous vous demandez peut-être s'il est possible d'arrondir une constante. Pour voir ce qui se passe, ajoutez temporairement la ligne suivante dans test.pl :

```
&Montest::arrondir(3);
```

Lancez "make test", et remarquez que Perl s'arrête avec une erreur fatale. Perl ne vous laisse pas modifier la valeur des constantes !

2.8 QU'Y A-T-IL DE NOUVEAU ICI ?

Il y a deux nouveautés ici. D'abord, nous avons apporté des modifications à Makefile.PL. En l'occurrence, nous avons spécifié une librairie supplémentaire à lier dans l'exécutable, la librairie mathématique libm. Nous expliquerons plus loin comment écrire des XSUB qui appellent une routine quelconque dans une librairie.

Ensuite, le résultat est renvoyé à l'appelant non par la valeur de retour de la fonction, mais par la variable qui a été passée à la fonction comme argument.

2.9 PARAMETRES D'ENTREE ET DE SORTIE

Vous précisez les paramètres à passer à la XSUB juste après avoir déclaré la valeur de retour et le nom de la fonction. Les blancs en début de ligne et le point-virgule à la fin sont facultatifs.

La liste des paramètres de sortie est spécifiée après la directive OUTPUT:. L'utilisation de RETVAL indique à Perl que cette valeur doit être renvoyée par la fonction XSUB. Dans l'exemple 3, comme la valeur que nous voulions renvoyer était contenue dans la variable passée en argument elle-même, nous avons rajouté cette variable (et non RETVAL) dans la section OUTPUT:.

2.10 LE COMPILATEUR XSUBPP

Le compilateur xsubpp prend le code XS dans le fichier .xs et le convertit en code C, en le mettant dans un fichier avec la terminaison .c. Le code C généré fait un grand usage des fonctions C internes à Perl.

2.11 LE FICHIER TYPEMAP

La compilateur xsubpp se base sur un ensemble de règles pour convertir les types de données Perl (scalaire, tableau, etc...) en types de données C (int, char *, etc...). Ces règles sont stockées dans le fichier typemap ($PERLLIB/ExtUtils/typemap), qui est divisé en trois parties.

La première partie tente de rapporter les différents types de données C à un code qui a une relation de correspondance avec les types Perl. La deuxième partie contient du code C utilisé par xsubpp pour les paramètres d'entrée. La troisième partie contient du code C utilisé par xsubpp pour les paramètres de sortie. Nous parlerons en détail du code C plus loin.

Considérons à présent une partie du fichier .c créé pour notre extension.

```
XS(XS_Montest_arrondir)
{
    dXSARGS;
    if (items != 1)
        croak("Usage: Montest::arrondir(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);
}
```

Remarquez les deux lignes marquées d'un "XXXXX". En consultant la première section du fichier typemap, vous constaterez que les doubles sont associés au type T_DOUBLE. Dans la section INPUT, un argument de type T_DOUBLE est assigné à une variable en lui appliquant la routine SvNV, puis en convertissant le résultat en double, et en l'assignant à la variable. De même, d'après la section OUTPUT, utilisée lorsque arg possède sa valeur finale, arg est passé à la fonction sv_setnv pour être renvoyé à la sous-routine appelante. Ces deux fonctions sont décrites dans *perlguts* ; nous préciserons plus loin, dans la section concernant la pile des arguments, ce que signifie ce "ST(0)".

2.12 AVERTISSEMENT

En général, c'est une mauvaise idée d'écrire des extensions qui modifient leurs paramètres d'entrée, comme dans l'exemple 3. Toutefois ce comportement est toléré pour permettre d'appeler de manière plus commode des routines C pré-existantes, lesquelles modifient fréquemment leurs paramètres d'entrée. L'exemple qui suit montre comme faire cela.

2.13 EXEMPLE 4

Dans cet exemple, nous commencerons à écrire des XSUB qui interagissent avec des librairies C prédéfinies. Construisons tout d'abord une petite librairie à nous, et laissons h2xs écrire à notre place les fichiers .pm et .xs.

Créez un nouveau répertoire nommé Montest2 au même niveau que le répertoire Montest. Sous Montest2, créez un autre répertoire nommé malib, et positionnez-vous dedans.

Nous allons créer à cet endroit quelques fichiers qui généreront une librairie de test. Parmi eux se trouvera un fichier-source en C et un fichier d'en-tête. Nous allons aussi créer un Makefile.PL dans ce répertoire. Ensuite nous nous assurerons que l'exécution de make au niveau Montest2 lance automatiquement l'exécution de ce fichier Makefile.Pl et du Makefile résultant.

Dans le répertoire malib, créez un fichier malib.h ressemblant à ceci :

```
#define TESTVAL 4

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

Créez aussi un fichier malib.c avec le contenu suivant :

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

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

Et créez enfin un fichier Makefile.PL avec ceci :

```
use ExtUtils::MakeMaker;
$Verbose = 1;
WriteMakefile(
    NAME      => 'Montest2::malib',
    SKIP      => [qw(all static static_lib dynamic dynamic_lib)],
    clean     => {'FILES' => 'libmalib$(LIB_EXT)'},
);

sub MY::top_targets {
        '
all :: static

static ::       libmalib$(LIB_EXT)

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

';
}
```

Nous allons maintenant créer les fichiers du répertoire principal Montest2. Allez dans le répertoire au-dessus de Montest2 et lancez la commande suivante :

```
% h2xs -O -n Montest2 ./Montest2/malib/malib.h
```

Vous recevrez un avertissement signalant que Montest2 est écrasé, mais c'est normal. Nos fichiers sont entreposés dans Montest2/malib, et ils ne seront pas modifiés.

Le Makefile.PL normal généré par h2xs ne connaît pas le répertoire malib. Nous devons lui dire qu'il y a un sous-répertoire dans lequel nous allons générer une librairie. Rajoutons le couple clé-valeur suivant dans l'appel de WriteMakefile :

```
'MYEXTLIB' => 'malib/libmalib$(LIB_EXT)',
```

ainsi qu'une nouvelle sous-routine de remplacement :

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

(Remarque : La plupart des make imposent que la ligne `cd malib && $(MAKE) $(PASTHRU)` soit indentée avec une tabulation, de même que pour le Makefile dans le sous-répertoire)

Adaptons aussi le fichier MANIFEST afin qu'il reflète correctement le contenu de notre extension. La ligne spécifiant "malib" devrait être remplacée par les trois lignes suivantes :

```
malib/Makefile.PL
malib/malib.c
malib/malib.h
```

Afin de conserver un espace de nommage cohérent et non pollué, ouvrez le fichier .pm et modifiez les lignes initialisant @EXPORT et @EXPORT_OK (il y en a deux : une sur la ligne commençant par "use vars" et l'autre lors de l'initialisation elle-même du tableau). Enfin, dans le fichier .xs, modifiez ainsi la ligne #include :

```
#include "malib/malib.h"
```

Et rajoutez aussi cette definition de fonction à la fin du fichier .xs :

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

Maintenant nous devons aussi créer un fichier typemap, car Perl, par défaut, ne supporte pas actuellement le type const char *. Créez un fichier nommé typemap et mettez-y la ligne suivante :

```
const char *    T_PV
```

A présent, lancez Perl sur le Makefile.PL au niveau le plus élevé. Remarquez qu'il crée aussi un Makefile dans le répertoire malib. Lancez make, et constatez qu'il va bien dans le répertoire malib pour y faire là aussi un make.

A présent, éditez le script test.pl et modifiez le block BEGIN pour afficher "1..4", et ajoutez les lignes suivantes à la fin du script :

```
print &Montest2::toto(1, 2, "Bonjour !") == 7 ? "ok 2\n" : "not ok 2\n";
print &Montest2::toto(1, 2, "0.0") == 7 ? "ok 3\n" : "not ok 3\n";
print abs(&Montest2::toto(0, 0, "-3.4") - 0.6) <= 0.01 ? "ok 4\n" : "not ok 4\n";
```

(Lorsqu'on fait des comparaisons en virgule flottante, il est souvent bon de ne pas vérifier l'égalité stricte, mais plutôt la différence en-dessous d'un certain facteur epsilon, ici 0,01).

Lancez "make test" et tout devrait être au point.

2.14 QUE S'EST-IL PASSE ICI ?

Contrairement aux exemples précédents, nous avons lancé h2xs sur un vrai fichier d'en-tête, ce qui a fait apparaître des éléments supplémentaires dans les fichiers .pm et .xs.

– Dans le fichier .xs, il y a maintenant une déclaration #include, avec le chemin complet du fichier d'en-tête malib.h.
– Un peu de code C a été rajouté dans le fichier .xs. La routine `constant` sert à rendre disponibles dans le script Perl (en l'occurrence, en appelant `&main::TESTVAL`) les valeurs définies par #define dans le fichier d'en-tête. Il y a aussi du code XS pour permettre l'appel de la routine `constant`.
– Le fichier .pm a exporté le nom TESTVAL dans le tableau @EXPORT. Cela pourrait entraîner des conflits de nommage. Un bon principe est que, si un identifiant spécifié par un #define ne doit être utilisé que par les routines C elles-mêmes, et non par l'utilisateur, alors il convient de l'enlever du tableau @EXPORT. D'autre part, si cela ne vous gêne pas d'utiliser le nom complet de la variable, vous pouvez supprimer la plupart ou la totalité des éléments du tableau @EXPORT.
– Si notre fichier d'en-tête contenait des directives #include, elles ne seraient pas prises en compte par h2xs. Il n'y a pas de solution satisfaisante pour l'instant.

Nous avons aussi indiqué à Perl la librairie construite dans le sous-répertoire malib. Il a suffi pour cela de rajouter la variable MYEXTLIB lors de l'appel de WriteMakefile et de réécrire la sous-routine postamble afin de lui faire exécuter make depuis le sous-répertoire. Le Makefile.PL associé à la librairie est un peu plus compliqué, mais sans excès. Nous avons là aussi remplacé la sous-routine postamble pour y insérer notre propre code. Ce code disait simplement que la librairie à créer devait être une archive statique (et non une librairie dynamique), et fournissait les commandes permettant de la construire.

2.15 LE PASSAGE D'ARGUMENTS A XSUBPP

Suite à l'exemple 4, il nous est désormais facile de simuler en Perl des librairies existantes dont les interfaces ne sont pas toujours de conception très rigoureuse. Poursuivons à présent par une discussion des arguments passés au compilateur xsubpp.

Lorsque vous spécifiez les arguments dans le fichier .xs, vous passez en réalité trois informations pour chacun d'entre eux. La première information est le rang de l'argument dans la liste (premier, second, etc...). La seconde est le type de l'argument ; il s'agit de sa déclaration de type (par exemple int, char*, etc...). La troisième information donne le mode de passage de l'argument lorsque la fonction de la librairie est appelée par cette XSUB. Il faut préciser si un "&" doit être placé devant l'argument, ce qui signifie que l'argument doit être passé comme une addresse sur le type de donnée spécifié.

Il y a une différence entre les deux arguments dans la fonction imaginaire suivante :

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

Le premier argument de cette fonction serait traité comme un char, il serait assigné à la variable a, et son adresse serait passée à la fonction toto. Le second argument serait traité comme un pointeur vers une chaîne de caractères, et assigné à la variable b. La *valeur* de b serait passée à la fonction toto. L'appel de la fonction toto qui serait effectivement généré par xsubpp ressemblerait à ceci :

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

xsubpp analysera de la même manière les listes d'arguments de fonction suivantes :

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

Toutefois, pour faciliter la compréhension, on conseille de placer un "&" près du nom de la variable et loin du type, et de placer un "*" près du type et loin du nom (comme c'est le cas dans l'exemple ci-dessus). Avec cette technique, il est facile de comprendre exactement ce qui sera passé à la fonction C – ce sera l'expression, quelle qu'elle soit, qui se trouve dans la "dernière colonne".

Vous devriez vraiment vous efforcer de passer à la fonction le type de variable qu'elle attend, lorsque c'est possible. Cela vous évitera de nombreux problèmes à long terme.

2.16 LA PILE DES ARGUMENTS

En regardant n'importe quel code généré par chacun des exemples, sauf le premier, vous remarquerez plusieurs références à ST(n), où n est la plupart du temps 0. "ST" est en fait une macro qui désigne le n-ième argument sur la pile. ST(0) est donc le premier argument passé à la XSUB, ST(1) le second, et ainsi de suite.

Quand vous faites la liste des arguments de la XSUB dans le fichier .xs, vous dites à xsubpp à quel argument correspond chaque élément de la pile (le premier dans la liste est le premier argument, et ainsi de suite). Vous vous exposez au pire si vous ne les énumérez pas dans l'ordre où la fonction les attend.

2.17 ETENDRE VOTRE EXTENSION

Parfois, vous souhaiterez peut-être fournir des méthodes ou des sous-routines supplémentaires pour vous permettre de rendre l'interface entre Perl et votre extension plus simple ou plus facile à comprendre. Ces routines devraient être implémentées dans le fichier .pm. C'est l'emplacement de leur définition à l'intérieur du fichier .pm qui détermine si elles sont chargées automatiquement en même temps que l'extension elle-même, ou seulement quand on les appelle.

2.18 DOCUMENTATION DE VOTRE EXTENSION

Aucune excuse ne peut vous dispenser de rédiger une documentation pour de votre extension. Sa place est dans le fichier .pm. Ce fichier sera passé à pod2man, et la documentation intégrée sera convertie au format manpage, puis placée dans le répertoire blib. Elle sera enfin copiée dans le répertoire man de Perl lors de l'installation de l'extension.

Vous pouvez entremêler la documentation et le code Perl dans votre fichier .pm. En fait, vous y serez forcé si vous comptez utiliser l'autochargement des méthodes, comme l'explique un commentaire dans le fichier .pm.

Pour plus d'informations sur le format pod, consultez *perlpod*.

2.19 INSTALLATION DE VOTRE EXTENSION

Une fois que votre extension est complète et qu'elle a passé tous ses tests avec succès, l'installation est plutôt simple : vous avez juste à lancer "make install". Vous devrez avoir des droits en écriture dans les répertoires où Perl est installé, ou bien vous devrez demander à votre administrateur système de lancer le make à votre place.

2.20 VOIR AUSSI

Pour plus d'informations, consultez *perlguts*, *perlxs*, *perlmod*, et *perlpod*.

3 AUTEUR

Jeff Okamoto <okamoto@corp.hp.com>

Revu et assisté par Dean Roehrich, Ilya Zakharevich, Andreas Koenig, et Tim Bunce.

3.1 Date de dernière modification

1996/7/10 pour la version originale

4 TRADUCTION

4.1 Version

Cette traduction française correspond à la version anglaise distribuée avec perl 5.005_02. Pour en savoir plus concernant ces traductions, consultez http://perl.enstimac.fr/.

4.2 Traducteur

Thierry Bézecourt <thbzcrt@mail.com>.

4.3 Relecture

Personne pour l'instant.

5 À propos de ce document

Ce document est la traduction française du document orginal distribué avec perl. Vous pouvez retrouver l'ensemble de la documentation française Perl (éventuellement mise à jour) en consultant l'URL <http://perl.enstimac.fr/>.

Ce document PDF a été produit Paul Gaborit. Si vous utilisez la version PDF de cette documentation (ou une version papier issue de la version PDF) pour tout autre usage qu'un usage personnel, je vous serai reconnaissant de m'en informer par un petit message <mailto:Paul.Gaborit@enstimac.fr>.

Si vous avez des remarques concernant ce document, en premier lieu, contactez la traducteur (vous devriez trouver son adresse électronique dans la rubrique TRADUCTION) et expliquez-lui gentiment vos remarques ou critiques. Il devrait normalement vous répondre et prendre en compte votre avis. En l'absence de réponse, vous pouvez éventuellement me contacter.

Vous pouvez aussi participer à l'effort de traduction de la documentation Perl. Toutes les bonnes volontés sont les bienvenues. Vous devriez trouver tous les renseignements nécessaires en consultant l'URL ci-dessus.