perlxs

Table des matières

1 NAME/NOM

perlxs - Manuel de référence du langage XS

2 DESCRIPTION

2.1 Introduction

XS est un langage utilisé pour réaliser une interface d'extension entre Perl et une librairie C qu'on souhaite utiliser depuis Perl. L'interface XS est combinée avec la librairie pour produire une nouvelle librairie susceptible d'être liée avec Perl. Une **XSUB** est une fonction écrite dans le langage XS ; c'est le composant central de l'interface de l'application Perl.

Le compilateur XS s'appelle **xsubpp**. Ce compilateur insère les constructions nécessaires pour permettre à une XSUB, qui n'est autre qu'une fonction C déguisée, de manipuler des valeurs Perl, et crée la colle nécessaire pour permettre à Perl d'accéder à la XSUB. Le compilateur utilise des **typemaps** pour faire la correspondance entre les variables et les paramètres de fonction C d'une part, et les valeurs Perl de l'autre. Le typemap par défaut gère de nombreux types C parmi les plus courants. Un typemap supplémentaire doit être créé pour traiter les structures et les types spécifiques à la librairie que l'on veut lier dans Perl.

Consultez *perlxstut* pour un guide d'apprentissage du processus de création d'extensions dans son ensemble.

Note : pour beaucoup d'extensions, le système SWIG de Dave Beazley fournit un mécanisme nettement plus pratique pour réaliser le code de liaison XS. Voyez http://www.cs.utah.edu/~beazley/SWIG pour plus d'information.

2.2 Mise en route

La plupart des exemples qui suivent portent sur la création d'une interface entre Perl et les fonctions de la librairie de connexion ONC+ RPC. La fonction rpcb_gettime() sera utilisée pour illustrer de nombreuses caractéristiques du langage XS. Cette fonction accepte deux paramètres ; le premier est une donnée en entrée et le second une donnée en sortie. La fonction renvoie aussi une valeur d'état.

```
bool_t rpcb_gettime(const char *host, time_t *timep);
```

Depuis C, cette fonction est appelée avec les instructions suivantes.

```
#include <rpc/rpc.h>
bool_t status;
time_t timep;
status = rpcb_gettime( "localhost", &timep );
```

Si on réalise une XSUB offrant une traduction directe entre cette fonction et Perl, cette XSUB sera utilisée depuis Perl avec le code suivant. Les variables $status et $timep contiendront la sortie de cette fonction.

```
use RPC;
$status = rpcb_gettime( "localhost", $timep );
```

Le fichier XS suivant présente une sous-routine XS, ou XSUB, proposant un exemple d'interface avec la fonction rpcb_gettime(). Cette XSUB représente une traduction directe entre C et Perl et préserve donc l'interface même depuis Perl. Cette XSUB sera appelée depuis Perl de la même manière que ci-dessus. Notez que les trois premières instructions #include, pour `EXTERN.h`, `perl.h` et `XSUB.h`, seront tout le temps présentes au début d'un fichier XS. On élargira plus tard cette approche, et on en présentera d'autres.

```
#include "EXTERN.h"
#include "perl.h"
#include "XSUB.h"
#include <rpc/rpc.h>

MODULE = RPC  PACKAGE = RPC

bool_t
rpcb_gettime(host,timep)
     char *host
     time_t &timep
     OUTPUT:
     timep
```

Toutes les extensions à Perl, y compris celles qui contiennent des XSUB, doivent être accompagnées d'un module Perl d'amorçage (angl. bootstrap) qui assure l'intégration de l'extension dans Perl. Ce module exporte les fonctions et les variables de l'extension vers le programme Perl et entraîne la liaison entre les XSUB de l'extension et Perl. Le module qui suit sera utilisé dans la plupart des exemples de ce document ; il doit être utilisé depuis Perl avec la commande `use` comme on l'a vu précédemment. On approfondira l'étude des modules Perl plus loin dans ce document.

```
package RPC;

require Exporter;
require DynaLoader;
@ISA = qw(Exporter DynaLoader);
@EXPORT = qw( rpcb_gettime );

bootstrap RPC;
1;
```

Tout au long de ce document, diverses interfaces à la XSUB rpcb_gettime() seront explorées. L'ordre ou le nombre des paramètres pris par les XSUB variera. Dans chaque cas, la XSUB est une abstraction entre Perl et la vraie fonction C rpcb_gettime(), et la XSUB doit toujours s'assurer que la vraie fonction rpcb_gettime() est appelée avec les bons paramètres. Cette abstraction permettra au programmeur de réaliser une interface avec la fonction C plus proche de Perl.

2.3 Anatomie d'une XSUB

La XSUB qui suit permet à un programme Perl d'accéder à une fonction de librairie C dont le nom est sin(). La XSUB fait comme la fonction C, qui prend un seul paramètre et renvoie une valeur unique.

```
double
sin(x)
  double x
```

Lorsqu'on utilise des pointeurs C, l'opérateur d'indirection `*` devrait être considéré comme une partie du type et l'opérateur d'addressage `&` comme une partie de la variable, comme on l'a fait dans la fonction rpcb_gettime() ci-dessus. Consultez la section sur les typemaps pour avoir plus de détails sur l'utilisation des qualificateurs et des opérateurs unaires dans les types C.

Le nom de la fonction et le type de retour doivent être placés sur des lignes séparées.

```
INCORRECT                       CORRECT
```

```
double sin(x)                   double
  double x                      sin(x)
                                  double x
```

Le corps de la fonction peut être mis en retrait ou ajusté à gauche. L'exemple suivant montre une fonction dont le corps est ajusté à gauche. On utilisera un retrait dans la plupart des exemples de ce document.

```
CORRECT

double
sin(x)
double
```

2.4 La pile des arguments

La pile des arguments est utilisée pour stocker les valeurs qui sont envoyées à la XSUB en tant que paramètres, ainsi que la valeur de retour de la XSUB. En fait, toutes les fonctions Perl mettent leurs valeurs sur cette pile en même temps, chacune étant limitée à son propre intervalle de positions sur la pile. Dans ce document, la première position dans la pile appartenant à la fonction active sera désignée comme la position 0 pour cette fonction.

Les XSUB se réfèrent à leurs arguments de la pile avec la macro **ST(x)**, où *x* désigne une position dans l'intervalle appartenant à la XSUB sur la pile. La position 0 pour cette fonction est connue de la XSUB comme ST(0). Les paramètres en entrée et les valeurs de retour de la XSUB commencent toujours à ST(0). Dans de nombreux cas simples, le compilateur **xsubpp** générera le code nécessaire à la manipulation de la pile des arguments en insérant des morceaux de code trouvés dans les typemaps. Dans les cas plus complexes, le programmeur devra fournir le code.

2.5 La variable RETVAL

La variable RETVAL est une variable magique qui est toujours du type retourné par la fonction de la librairie C. Le compilateur **xsubpp** fournit cette variable dans chaque XSUB et l'utilise par défaut pour y mettre la valeur de retour de la fonction C appelée. Dans les cas simples, la valeur de RETVAL sera mise dans ST(0) sur la pile d'arguments, où Perl le recevra comme valeur de retour de la XSUB.

Si la XSUB a un type de retour égal à `void`, le compilateur ne fournira pas de variable RETVAL pour cette fonction. Lorsqu'on utilise la directive PPCODE:, la variable RETVAL n'est plus nécessaire, sauf si on l'utilise explicitement.

Si la directive PPCODE: n'est pas utilisée, la valeur de retour `void` devrait être utilisée uniquement pour des sous-routines qui ne renvoient pas de valeur, *même si* la directive CODE: est utilisée pour positionner ST(0) explicitement.

Dans des versions plus anciennes de ce document, on conseillait d'utiliser la valeur de retour `void` dans de tels cas. On a découvert que cela pouvait mener à des segfaults lorsque la XSUB était *vraiment* `void`. Cette pratique est maintenant désapprouvée, et risque de ne plus être supportée dans une version future. Utilisez la valeur de retour `SV *` dans ces cas-là (`xsubpp` contient actuellement du code heuristique qui tente de faire la différence entre les fonctions "vraiment-void" et celles qui sont "déclarées-void-suivant-l'ancienne-pratique" ; votre code est donc à la merci de l'heuristique si vous n'utilisez pas `SV *` comme valeur de retour).

2.6 Le mot-clé MODULE

Le mot-clé MODULE est utilisé pour marquer le début du code XS et spécifier un paquetage pour les fonctions que l'on est en train de définir. Tout le texte qui précède le premier mot-clé MODULE est considéré comme du code C et sera transféré dans la sortie du compilateur sans modification. Tout module XS aura une fonction d'initialisation utilisée pour brancher la XSUB dans Perl. Le nom du paquetage de cette fonction d'initialisation correspondra à la valeur de la dernière instruction MODULE dans les fichiers source XS. La valeur de MODULE devrait toujours rester constante à l'intérieur d'un même fichier XS, même si cela n'est pas obligatoire.

L'exemple suivant démarre le code XS et place toutes les fonctions dans un paquetage nommé RPC.

```
    MODULE = RPC
```

2.7 Le mot-clé PACKAGE

Lorsque les fonctions, à l'intérieur d'un fichier source XS, doivent être réparties dans des paquetages, il faut utiliser le mot-clé PACKAGE. Ce mot-clé est utilisé avec le mot-clé MODULE et doit être spécifié immédiatement après lui.

```
MODULE = RPC  PACKAGE = RPC

[ code XS dans le paquetage RPC ]

MODULE = RPC  PACKAGE = RPCB

[ code XS dans le paquetage RPCB ]

MODULE = RPC  PACKAGE = RPC

[ code XS dans le paquetage RPC ]
```

Bien que ce mot-clé soit optionnel et qu'il fournisse des informations redondantes dans certains cas, il devrait toujours être utilisé. Il permet de garantir que la XSUB apparaîtra dans le paquetage désiré.

2.8 Le mot-clé PREFIX

Le mot-clé PREFIX désigne des préfixes à supprimer dans les noms de fonction Perl. Si la fonction C est `rpcb_gettime()` et que la valeur de PREFIX est `rpcb_`, Perl devra voir cette fonction comme `gettime()`.

Ce mot-clé suit le mot-clé PACKAGE lorsqu'il est utilisé. Si PACKAGE n'est pas utilisé, alors PREFIX suit le mot-clé MODULE.

```
MODULE = RPC  PREFIX = rpc_

MODULE = RPC  PACKAGE = RPCB  PREFIX = rpcb_
```

2.9 Le mot-clé OUTPUT

Le mot-clé OUTPUT: indique qu'il faut mettre à jour certains paramètres de fonction (en rendant les nouvelles valeurs accessibles à Perl) lorsque la XSUB se termine, ou que certaines valeurs doivent être renvoyées à la fonction Perl. Pour des fonctions simples, comme la fonction sin() ci-dessus, la variable RETVAL est désignée automatiquement comme une valeur en sortie. Dans des fonctions plus complexes, le compilateur **xsubpp** aura besoin d'aide pour déterminer quelles variables sont des variables en sortie.

Ce mot-clé sera utilisé normalement en complément du mot-clé CODE:. La variable RETVAL n'est pas reconnue comme une variable en sortie lorsque le mot-clé CODE: est présent. Le mot-clé OUTPUT: est utilisé dans cette situation pour dire au compilateur que RETVAL est réellement une variable en sortie.

Le mot-clé OUTPUT: peut aussi être utilisé pour indiquer que des paramètres de fonction sont des variables en sortie. Cela peut être nécessaire lorsqu'un paramètre a été modifié à l'intérieur d'une fonction et que le programmeur veut que cette modification soit visible depuis Perl.

```
bool_t
rpcb_gettime(host,timep)
     char *host
     time_t &timep
     OUTPUT:
     timep
```

Le mot-clé OUTPUT: permet aussi à un paramètre en sortie d'être relié à un morceau de code correspondant plutôt qu'à un typemap.

```
bool_t
rpcb_gettime(host,timep)
      char *host
      time_t &timep
      OUTPUT:
      timep sv_setnv(ST(1), (double)timep);
```

xsubpp rajoute automatiquement un `SvSETMAGIC()` pour tous les paramètres de la section OUTPUT de la XSUB, sauf pour RETVAL. C'est le comportement désiré la plupart du temps, car il se charge d'invoquer convenablement l'effet 'set' (angl. "set magic") sur les données en sortie (ce qui est nécessaire pour les éléments de tableau simple ou de tableau associatif passés en paramètre, qui doivent être créés s'ils n'existent pas). Si, pour une raison donnée, ce comportement n'est pas désiré, la section OUTPUT peut contenir une ligne `SETMAGIC: DISABLE` qui le désactive pour le restant des paramètres de la section OUTPUT:. De même, `SETMAGIC: ENABLE` peut être utilisé pour le réactiver pour le restant de la section OUTPUT. Voyez *perlguts* pour plus de détails sur l'effet 'set'.

2.10 Le mot-clé CODE

Ce mot-clé est utilisé dans des XSUB plus complexes qui nécessitent un traitement spécial pour la fonction C. La variable RETVAL est disponible, mais elle ne sera pas retournée, sauf si elle est spécifiée sous le mot-clé OUTPUT:.

La XSUB qui suit correspond à une fonction C qui requiert un traitement spécial de ses paramètres. L'utilisation depuis Perl est donnée en premier.

```
$status = rpcb_gettime( "localhost", $timep );
```

Voici la XSUB.

```
bool_t
rpcb_gettime(host,timep)
      char *host
      time_t timep
      CODE:
           RETVAL = rpcb_gettime( host, &timep );
      OUTPUT:
      timep
      RETVAL
```

2.11 Le mot-clé INIT

Le mot-clé INIT: permet d'insérer du code d'initialisation dans la XSUB avant que le compilateur ne génère l'appel à la fonction C. Contrairement au mot-clé CODE: ci-dessus, celui-ci n'affecte pas la manière dont le compilateur traite RETVAL.

```
bool_t
rpcb_gettime(host,timep)
      char *host
      time_t &timep
      INIT:
      printf("# l'hote est %s\n", host );
      OUTPUT:
      timep
```

2.12 Le mot-clé NO_INIT

Le mot-clé NO-INIT sert à indiquer qu'on utilise un paramètre de la fonction uniquement comme valeur en sortie. Le compilateur **xsubpp** génère normalement du code pour lire les valeurs de tous les paramètres de la fonction sur la pile des arguments, et pour les assigner à des variables C lors de l'entrée dans la fonction. NO_INIT dit au compilateur que certains paramètres seront utilisés en sortie plutôt qu'en entrée, et qu'ils seront traités avant la fin de la fonction.

L'exemple suivant présente une variante de la fonction rpcb_gettime(). Cette fonction utilise la variable timep uniquement comme variable en sortie, et ne se préoccupe pas de son contenu initial.

```
bool_t
rpcb_gettime(host,timep)
     char *host
     time_t &timep = NO_INIT
     OUTPUT:
     timep
```

2.13 Initialisation des arguments de fonction

Normalement, les paramètres de fonction sont initialisés avec les valeurs qu'ils ont dans la pile des arguments. Les typemaps contiennent les portions de code utilisées pour transférer les valeurs Perl dans des paramètres C. Le programmeur, toutefois, est autorisé à surcharger les typemaps et à fournir un code d'initialisation différent (ou supplémentaire).

Le code qui suit montre comment fournir du code d'initialisation pour les paramètres de fonction. Le code d'initialisation est évalué entre des guillemets par le compilateur (en utilisant eval()), avant d'être inséré dans le code généré, de sorte que toute expression devant être interprétée littéralement (essentiellement `$`, `@` ou \) doit être protégée par des \. Les variables `$var`, `$arg` et `$type` peuvent être utilisées comme dans les typemaps.

```
bool_t
rpcb_gettime(host,timep)
     char *host = (char *)SvPV($arg,PL_na);
     time_t &timep = 0;
     OUTPUT:
     timep
```

Ce procédé ne devrait pas être utilisé pour fournir des valeurs par défaut aux paramètres. Le cas normal d'utilisation est celui où un paramètre de fonction doit être traité par une autre fonction de la librairie avant d'être utilisé. Les paramètres par défaut font l'objet de la prochaine section.

Si l'initialisation commence par =, elle sera insérée sur la même ligne que la déclaration de la variable. Si elle commence par ; ou +, elle sera insérée une fois que toutes les variables en entrée auront été déclarées. Les cas = et ; remplacent l'initialisation fournie normalement par le typemap. Dans le cas +, l'initialisation du typemap précédera le code d'initialisation qui suit le +. Une variable globale, `%v`, est disponible pour le cas vraiment rare où une initialisation a besoin d'une information venant d'une autre initialisation.

```
bool_t
rpcb_gettime(host,timep)
     time_t &timep ; /*\$v{time}=@{[$v{time}=$arg]}*/
     char *host + SvOK($v{time}) ? SvPV($arg,PL_na) : NULL;
     OUTPUT:
     timep
```

2.14 Valeurs de paramètres par défaut

On peut donner des valeurs par défaut à des paramètres de fonction en rajoutant une instruction d'affectation dans la liste des paramètres. La valeur par défaut peut être un nombre ou une chaîne de caractères. Les valeurs par défaut doivent toujours être utilisées uniquement pour les paramètres les plus à droite.

Afin de permettre à la XSUB de rpcb_gettime() d'avoir un hôte par défaut, les paramètres de la XSUB pourraient être réordonnés. La XSUB appellera alors la vraie fonction rpcb_gettime() avec les paramètres placés dans le bon ordre. Perl appellera cette XSUB avec l'une des instructions suivantes.

```
$status = rpcb_gettime( $timep, $host );

$status = rpcb_gettime( $timep );
```

La XSUB ressemblera au code qui suit. Un bloc CODE: est utilisé pour appeler la vraie fonction rpcb_gettime() avec les paramètres remis dans l'ordre correct pour cette fonction.

```
bool_t
rpcb_gettime(timep,host="localhost")
     char *host
     time_t timep = NO_INIT
     CODE:
          RETVAL = rpcb_gettime( host, &timep );
     OUTPUT:
     timep
     RETVAL
```

2.15 Le mot-clé PREINIT

Le mot-clé PREINIT: permet de déclarer des variables supplémentaires avant que les typemaps soient utilisés. Si une variable est déclarée dans un bloc CODE:, elle suivra tout code généré par le typemap. Cela peut provoquer des fautes de syntaxe en C. Pour forcer la déclaration de la variable avant le code du typemap, placez-la dans un bloc PREINIT:. Le mot-clé PREINIT: peut être utilisé une ou plusieurs fois à l'intérieur d'une XSUB.

Les exemples qui suivent sont équivalents, mais, si le code utilise des typemaps complexes, le premier exemple sera plus sûr.

```
bool_t
rpcb_gettime(timep)
     time_t timep = NO_INIT
     PREINIT:
     char *host = "localhost";
     CODE:
     RETVAL = rpcb_gettime( host, &timep );
     OUTPUT:
     timep
     RETVAL
```

Un exemple correct, mais avec des risques d'erreur.

```
bool_t
rpcb_gettime(timep)
     time_t timep = NO_INIT
     CODE:
     char *host = "localhost";
     RETVAL = rpcb_gettime( host, &timep );
     OUTPUT:
     timep
     RETVAL
```

2.16 Le mot-clé SCOPE

Le mot-clé SCOPE: permet d'activer un niveau spécial de visibilité (angl. scoping) pour une XSUB donnée. Dans ce cas-là, la XSUB invoquera ENTER et LEAVE automatiquement (Note du Traducteur : les macros ENTER et LEAVE sont décrites dans *perlcall*).

Afin de supporter des correspondances de type complexes, ce niveau spécial sera activé automatiquement pour une XSUB si elle utilise une entrée de typemap contenant le commentaire `/*scope*/`.

Pour activer ce niveau spécial :

```
SCOPE: ENABLE
```

Pour le désactiver :

```
SCOPE: DISABLE
```

2.17 Le mot-clé INPUT

Habituellement, les paramètres de XSUB sont évalués juste après l'entrée dans la XSUB. Le mot-clé INPUT: peut être utilisé pour forcer ces paramètres à être évalués un peu plus tard. On peut utiliser le mot-clé INPUT: plusieurs fois dans une XSUB, et ceci pour une ou plusieurs variables en entrée. Ce mot-clé accompagne le mot-clé PREINIT:.

L'exemple suivant montre comment l'évaluation du paramètre en entrée `timep` peut être retardée, après un PREINIT.

```
bool_t
rpcb_gettime(host,timep)
      char *host
      PREINIT:
      time_t tt;
      INPUT:
      time_t timep
      CODE:
           RETVAL = rpcb_gettime( host, &tt );
           timep = tt;
      OUTPUT:
      timep
      RETVAL
```

Dans cet autre exemple, chacun des paramètres en entrée voit son évaluation retardée.

```
bool_t
rpcb_gettime(host,timep)
      PREINIT:
      time_t tt;
      INPUT:
      char *host
      PREINIT:
      char *h;
      INPUT:
      time_t timep
      CODE:
           h = host;
           RETVAL = rpcb_gettime( h, &tt );
           timep = tt;
      OUTPUT:
      timep
      RETVAL
```

2.18 Listes de paramètres de longueur variable

Les XSUB peuvent recevoir des listes de paramètres de longueur variable en spécifiant une ellipse `...` dans la liste des paramètres. Cette utilisation de l'ellipse est similaire à celle que l'on trouve en C ANSI. Le programmeur peut déterminer le nombre d'arguments passés à la XSUB en examinant la variable `items` que le compilateur **xsubpp** fournit pour chaque XSUB. Grâce à ce mécanisme, on peut réaliser une XSUB qui accepte une liste de paramètres de longueur indéterminée.

Le paramètre *host* de la XSUB rpcb_gettime() peut être optionnel, de manière à ce qu'on puisse utiliser l'ellipse pour indiquer que la XSUB prendra un nombre variable de paramètres. Perl devrait pouvoir appeler cette XSUB avec chacune des instructions suivantes.

```
$status = rpcb_gettime( $timep, $host );

$status = rpcb_gettime( $timep );
```

Voici le code XS, avec l'ellipse :

```
bool_t
rpcb_gettime(timep, ...)
     time_t timep = NO_INIT
     PREINIT:
     char *host = "localhost";
     CODE:
             if( items > 1 )
                  host = (char *)SvPV(ST(1), PL_na);
             RETVAL = rpcb_gettime( host, &timep );
     OUTPUT:
     timep
     RETVAL
```

2.19 Le mot-clé C_ARGS

Le mot-clé C_ARGS permet de réaliser des XSUB que l'on n'appelle pas de la même manière depuis Perl que depuis C, sans qu'il soit nécessaire d'écrire une section CODE: ou PPCODE:. Le contenu du paragraphe C_ARGS est passé comme argument à la fonction C, sans aucun changement.

Supposons par exemple que la fonction C soit déclarée ainsi :

```
symbolic nth_derivative(int n, symbolic function, int flags);
```

et que la valeur par défaut de *flags* soit contenue dans la variable C *default_flags*. Supposons que vous souhaitiez réaliser une interface que l'on appellera de la manière suivante :

```
$second_deriv = $function->nth_derivative(2);
```

Pour cela, déclarez la XSUB ainsi :

```
symbolic
nth_derivative(function, n)
    symbolic        function
    int             n
C_ARGS:
    n, function, default_flags
```

2.20 Le mot-clé PPCODE

Le mot-clé PPCODE: est une variante du mot-clé CODE: qui est utilisée pour indiquer au compilateur **xsubpp** que le programmeur fournit le code contrôlant la pile des arguments pour les valeurs de retour des XSUB. On souhaite parfois que la XSUB renvoie une liste de valeurs et non une valeur unique. Dans ce cas-là, il faut utiliser PPCODE: et rajouter de manière explicite la liste des valeurs sur la pile. Les mots-clés PPCODE: et CODE: ne sont pas utilisés simultanément dans la même XSUB.

La XSUB suivante appelle la fonction C rpcb_gettime() et renvoie à Perl ses deux valeurs de sortie, timep et status, comme une seule liste.

```
void
rpcb_gettime(host)
     char *host
     PREINIT:
     time_t  timep;
     bool_t  status;
     PPCODE:
     status = rpcb_gettime( host, &timep );
     EXTEND(SP, 2);
     PUSHs(sv_2mortal(newSViv(status)));
     PUSHs(sv_2mortal(newSViv(timep)));
```

Remarquez que le programmeur doit fournir le code C assurant l'appel de la vraie fonction rpcb_gettime(), ainsi que le placement correct des valeurs de retour sur la pile des arguments.

Le type de retour `void` pour cette fonction indique au compilateur **xsubpp** que la variable RETVAL n'est pas nécessaire, qu'elle n'est pas utlisée, et qu'elle ne devrait pas être créée. Dans la plupart des cas, il faut utiliser le type de retour void avec l'instruction PPCODE:.

On utilise la macro EXTEND() afin de dégager de la place sur la pile des arguments pour les 2 valeurs de retour. L'instruction PPCODE: fait en sorte que le compilateur **xsubpp** crée un pointeur vers la pile dans la variable `SP` ; c'est ce pointeur qui est utilisé dans la macro EXTEND(). Les valeurs sont ensuite rajoutées sur la pile avec les macros PUSHs().

A présent, la fonction rpcb_gettime() peut être utilisée depuis Perl avec l'instruction suivante.

```
($status, $timep) = rpcb_gettime("localhost");
```

Lorsque vous travaillez sur les paramètres en sortie avec une section PPCODE:, assurez-vous de traiter l'effet 'set' convenablement. Consultez *perlguts* pour plus de détails sur cet effet 'set'.

2.21 Renvoyer undef et des listes vides

Le programmeur voudra parfois renvoyer simplement `undef` ou une liste vide si la fonction échoue, plutôt qu'une valeur d'état à part. La fonction rpcb_gettime() présente justement cette situation. Nous aimerions que la fonction retourne l'heure si elle réussit, ou undef si elle échoue. Dans le code Perl suivant, la valeur de $timep sera soit undef, soit une heure valide;

```
$timep = rpcb_gettime( "localhost" );
```

La XSUB suivante n'utilise le type de retour `SV *` qu'à titre d'aide-mémoire, et il utilise un bloc CODE: pour indiquer au compilateur que le programmeur a fourni tout le code nécessaire. L'appel de sv_newmortal() initialise la valeur de retour à undef, ce qui en fait la valeur de retour par défaut.

```
SV *
rpcb_gettime(host)
     char *  host
     PREINIT:
     time_t  timep;
     bool_t x;
     CODE:
     ST(0) = sv_newmortal();
     if( rpcb_gettime( host, &timep ) )
          sv_setnv( ST(0), (double)timep);
```

Cet autre exemple montre comment on peut mettre un undef explicite dans la valeur de retour, au cas où le besoin s'en ferait ressentir.

```
SV *
rpcb_gettime(host)
     char *  host
     PREINIT:
     time_t  timep;
     bool_t x;
     CODE:
     ST(0) = sv_newmortal();
     if( rpcb_gettime( host, &timep ) ){
          sv_setnv( ST(0), (double)timep);
     }
     else{
          ST(0) = &PL_sv_undef;
     }
```

Pour renvoyer une liste vide, il faut utiliser un bloc PPCODE: et ne pas rajouter de valeur de retour sur la pile.

```
void
rpcb_gettime(host)
     char *host
     PREINIT:
     time_t  timep;
     PPCODE:
     if( rpcb_gettime( host, &timep ) )
          PUSHs(sv_2mortal(newSViv(timep)));
     else{
     /* Rien n'est remis sur la pile, donc une */
     /* liste vide est renvoyee implicitement */
     }
```

Certaines personnes préfèrent rajouter un `return` explicite dans la XSUB ci-dessus au lieu de laisser l'exécution se poursuivre jusqu'au bout. Dans ce cas-là, il convient plutôt d'utiliser `XSRETURN_EMPTY`, ce qui garantira que la pile XSUB est ajustée correctement. D'autres macros sont décrites dans *perlguts*.

2.22 Le mot-clé REQUIRE

Le mot-clé REQUIRE: sert à indiquer le numéro de version minimal du compilateur **xsubpp** requis pour compiler le module XS. Un module XS contenant l'instruction suivante ne pourra être compilé qu'avec une version de **xsubpp** égale ou supérieure à 1.922 :

```
REQUIRE: 1.922
```

2.23 Le mot-clé CLEANUP

Ce mot-clé peut être utilisé quand une XSUB doit exécuter des procédures de nettoyage spéciales avant de se terminer. Quand le mot-clé CLEANUP: est utilisé, il doit suivre tout bloc CODE:, PPCODE: ou OUTPUT: présent dans la XSUB. Les instructions spécifiées dans le bloc de nettoyage seront les dernières instructions de la XSUB.

2.24 Le mot-clé BOOT

Le mot-clé BOOT: permet de rajouter du code dans la fonction d'amorçage de l'extension. La fonction d'amorçage est générée par le compilateur **xsubpp** et contient normalement les instructions nécessaires pour enregistrer toute XSUB auprès de Perl. Avec le mot-clé BOOT:, le programmeur peut dire au compilateur de rajouter des instructions supplémentaires dans la fonction d'amorçage.

Ce mot-clé peut être utilisé n'importe quand après le premier mot-clé MODULE, et doit être tout seul sur une ligne. La première ligne blanche après le mot-clé terminera le bloc de code.

```
BOOT:
# Le message suivant sera affiche lors de l'execution de la
# fonction d'amorcage.
printf("bonjour !\n");
```

2.25 Le mot-clé VERSIONCHECK

Le mot-clé VERSIONCHECK: correspond aux options `-versioncheck` et `-noversioncheck` de **xsubpp**. Ce mot-clé prime sur les options de la ligne de commande. La vérification de version est activée par défaut. Lorsqu'elle est activée, le module XS essaie de vérifier que son numéro de version est compatible avec celui du module PM.

Pour activer la vérification de version :

```
VERSIONCHECK: ENABLE
```

Pour la désactiver :

```
VERSIONCHECK: DISABLE
```

2.26 Le mot-clé PROTOTYPES

Le mot-clé PROTOTYPES: correspond aux options `-prototypes` et `-noprototypes` de **xsubpp**. Ce mot-clé prime sur les options de la ligne de commande. Les prototypes sont activés par défaut. Lorsqu'ils sont activés, les XSUB reçoivent des prototypes Perl. Ce mot-clé peut être utilisé plusieurs fois dans un module XS pour activer et désactiver les prototypes dans différentes parties du module.

Pour activer les prototypes :

```
PROTOTYPES: ENABLE
```

Pour les désactiver :

```
PROTOTYPES: DISABLE
```

2.27 Le mot-clé PROTOTYPE

Ce mot-clé est proche du mot-clé PROTOTYPES: ci-dessus, mais peut être utilisé pour forcer **xsubpp** à utiliser un prototype donné pour la XSUB. Ce mot-clé prime sur toute autre option ou mot-clé relatif au prototypage, mais n'affecte que la XSUB courante. Consultez Prototypes in *perlsub* pour plus d'informations sur les prototypes Perl.

```
bool_t
rpcb_gettime(timep, ...)
      time_t timep = NO_INIT
      PROTOTYPE: $;$
      PREINIT:
      char *host = "localhost";
      CODE:
             if( items > 1 )
                  host = (char *)SvPV(ST(1), PL_na);
             RETVAL = rpcb_gettime( host, &timep );
      OUTPUT:
      timep
      RETVAL
```

2.28 Le mot-clé ALIAS

Le mot-clé ALIAS: permet à une XSUB de recevoir plusieurs noms en Perl, et de déterminer lequel de ces noms a été utilisé pour l'invoquer. Les noms en Perl peuvent être complets avec le nom du paquetage. Chaque alias reçoit un numéro. Le compilateur crée une variable nommée `ix` qui contient le numéro de l'alias utilisé. Lorsque le nom utilisé pour invoquer la XSUB est celui avec lequel il a été déclaré, `ix` est égal à 0.

L'exemple suivant crée des alias `FOO::gettime()` et `BAR::gettit()` pour cette fonction.

```
bool_t
rpcb_gettime(host,timep)
      char *host
      time_t &timep
      ALIAS:
        FOO::gettime = 1
        BAR::getit = 2
      INIT:
      printf("# ix = %d\n", ix );
      OUTPUT:
      timep
```

2.29 Le mot-clé INTERFACE

Ce mot-clé déclare que la XSUB courante sert à garder en réserve la signature de fonction indiquée. S'il est suivi par du texte, ce texte est considéré comme une liste de fonctions qui ont cette signature, et qui doivent être attachées à des XSUB.

Par exemple, si vous avez 4 fonctions multiply(), divide(), add() et subtract(), toutes avec la signature

```
symbolic f(symbolic, symbolic);
```

vous pouvez les implémenter toutes à la fois avec la XSUB

```
symbolic
interface_s_ss(arg1, arg2)
    symbolic        arg1
    symbolic        arg2
INTERFACE:
    multiply divide
    add subtract
```

L'avantage de cette approche par rapport au mot-clé ALIAS: est que l'on peut attacher une fonction supplémentaire remainder() lors de l'exécution en utilisant

```
CV *mycv = newXSproto("Symbolic::remainder",
                      XS_Symbolic_interface_s_ss, __FILE__, "$$");
XSINTERFACE_FUNC_SET(mycv, remainder);
```

(on suppose dans cet exemple qu'il n'y a pas de section INTERFACE_MACRO:, car il faut alors utiliser autre chose que `XSINTERFACE_FUNC_SET`)

2.30 Le mot-clé INTERFACE_MACRO

Ce mot-clé permet de définir une INTERFACE qui procède d'une manière différente pour récupérer dans la XSUB un pointeur vers la fonction C. Le texte qui suit ce mot-clé doit indiquer des noms de macros à utiliser pour récupérer le pointeur de fonction ou pour l'initialiser dans la XSUB. La macro de récupération reçoit le type de retour, `CV*`, et `XSANY.any_dptr` pour ce `CV*`. La macro d'initialisation reçoit cv et le pointeur de fonction.

Les valeurs par défaut de ces macros sont `XSINTERFACE_FUNC` et `XSINTERFACE_FUNC_SET`. Le mot-clé INTERFACE avec une liste vide de fonctions peut être omis si INTERFACE_MACRO est utilisé.

Supposons que, dans l'exemple qui précède, des pointeurs de fonction pour multiply(), divide(), add(), et subtract() soient conservés dans un tableau C global `fp[]`, les positions de ces pointeurs par rapport au début du tableau étant contenues dans des variables nommées `multiply_off`, `divide_off`, `add_off`, `subtract_off`. On peut alors utiliser

```
#define XSINTERFACE_FUNC_BYOFFSET(ret,cv,f) \
    ((XSINTERFACE_CVT(ret,))fp[CvXSUBANY(cv).any_i32])
#define XSINTERFACE_FUNC_BYOFFSET_set(cv,f) \
    CvXSUBANY(cv).any_i32 = CAT2( f, _off )
```

dans la section C, et

```
symbolic
interface_s_ss(arg1, arg2)
    symbolic        arg1
    symbolic        arg2
INTERFACE_MACRO:
    XSINTERFACE_FUNC_BYOFFSET
    XSINTERFACE_FUNC_BYOFFSET_set
INTERFACE:
    multiply divide
    add subtract
```

dans la section XSUB.

2.31 Le mot-clé INCLUDE

Ce mot-clé peut être utilisé pour insérer d'autres fichiers dans le module XS. Les autres fichiers peuvent contenir du code XS. INCLUDE: peut aussi être utilisé pour exécuter une commande générant du code XS à insérer dans le module.

Le fichier *Rpcb1.xsh* contient notre fonction `rpcb_gettime()` :

```
bool_t
rpcb_gettime(host,timep)
      char *host
      time_t &timep
      OUTPUT:
      timep
```

Le module XS peut utiliser INCLUDE: pour insérer ce fichier.

```
INCLUDE: Rpcb1.xsh
```

Si les paramètres du mot-clé INCLUDE: sont suivis d'un trait vertical (|), alors le compilateur interprétera les paramètres comme une commande.

```
INCLUDE: cat Rpcb1.xsh |
```

2.32 Le mot-clé CASE

Le mot-clé CASE permet à une XSUB de se subdiviser en plusieurs parties distinctes, chacune se comportant comme une XSUB virtuelle. CASE: est global à la XSUB : quand on l'utilise, il faut mettre tous les autres mots-clés XS à l'intérieur d'un CASE:. Cela signifie que rien ne peut précéder le premier CASE: dans la XSUB, et que tout code suivant le dernier CASE: y est incorporé.

Un CASE: peut être associé à une condition sur un paramètre de la XSUB en utilisant la variable-alias `ix` (voir Le mot-clé ALIAS: (§**??**)), ou éventuellement la variable `items` (voir Listes de paramètres de longueur variable (§2.18)). Le dernier CASE correspond au mot-clé **default** en C s'il n'est associé à aucune condition. L'exemple suivant montre un CASE dépendant de `ix` dans la fonction `rpcb_gettime()`, laquelle a pour alias `x_gettime()`. Lorsque la fonction est invoquée sous le nom `rpcb_gettime()`, elle reçoit les paramètres habituels `(char *host, time_t *timep)`, tandis que, lorsque elle est invoquée en tant que `x_gettime()`, ses paramètres sont inversés, `(time_t *timep, char *host)`.

```
long
rpcb_gettime(a,b)
  CASE: ix == 1
      ALIAS:
      x_gettime = 1
      INPUT:
      # 'a' est timep, 'b' est host
      char *b
      time_t a = NO_INIT
      CODE:
           RETVAL = rpcb_gettime( b, &a );
      OUTPUT:
      a
      RETVAL
  CASE:
      # 'a' est host, 'b' est timep
      char *a
      time_t &b = NO_INIT
      OUTPUT:
      b
      RETVAL
```

Cette fonction peut être appelée avec chacune des instructions suivantes. Notez la différence dans l'ordre des arguments.

```
    $status = rpcb_gettime( $host, $timep );

    $status = x_gettime( $timep, $host );
```

2.33 L'opérateur unaire &

L'opérateur unaire & est utilisé pour dire au compilateur qu'il doit déréférencer l'objet lors de l'appel à la fonction C. On utilise ce procédé lorsqu'on n'a pas mis de bloc CODE: et que l'objet n'est pas de type pointeur (l'objet est un `int` ou un `long`, mais pas un `int*` ni un `long*`).

La XSUB suivante génère du code C incorrect. Le code généré par **xsubpp** appellera `rpcb_gettime()` avec les paramètres `(char *host, time_t timep)`, mais le vrai `rpcb_gettime()` attend un paramètre `timep` de type `time_t *` et non `time_t`.

```
bool_t
rpcb_gettime(host,timep)
      char *host
      time_t timep
      OUTPUT:
      timep
```

On résout ce problème en utilisant l'opérateur `&`. Le compilateur **xsubpp**, à présent, traduira la XSUB en code qui appellera `rpcb_gettime()` de manière correcte, avec les paramètres `(char *host, time_t *timep)`. Il le fait en conservant le `&` tel quel, de sorte que l'appel de fonction est `rpcb_gettime(host, &timep)`.

```
bool_t
rpcb_gettime(host,timep)
      char *host
      time_t &timep
      OUTPUT:
      timep
```

2.34 Insérer des commentaires et des directives de pré-processeur C

Des directives de pré-processeur C peuvent prendre place à l'intérieur des blocs BOOT:, PREINIT:, INIT:, CODE:, PPCODE: et CLEANUP:, de même qu'en dehors des fonctions. Les commentaires sont autorisés partout après le mot-clé MODULE. Le compilateur transmet les directives de pré-processeur sans modification et supprime les lignes commentées.

On peut rajouter des commentaires dans les XSUB en mettant un `#` sur une ligne comme premier caractère non blanc. Attention, le commentaire ne doit pas ressembler à une directive de pré-processeur C, sinon il sera interprété comme tel. Le moyen le plus simple d'éviter cela consiste à mettre des caractères blancs devant le `#`.

Si vous utilisez des directives de pré-processeur pour choisir entre deux versions d'une fonction, utilisez

```
#if ... version1
#else /* ... version2  */
#endif
```

et non pas

```
#if ... version1
#endif
#if ... version2
#endif
```

car, dans le second cas, xsubpp croira que vous avez défini la fonction deux fois. Par ailleurs, insérez une ligne blanche avant le `#else` et le `#endif` afin qu'ils ne soient pas considérés comme une partie intégrante du corps de la fonction.

2.35 Utiliser XS avec C++

Si une fonction est définie comme une méthode C++, alors elle considère son premier argument comme un pointeur vers un objet. Le pointeur vers l'objet est stocké dans une variable nommée THIS. L'objet doit avoir été créé en C++ avec la fonction new(), et il doit être béni par Perl (comme avec la fonction bless() en Perl pur) avec la macro sv_setref_pv(). La *bénédiction* d'un objet par Perl peut être effectuée par le typemap. Un exemple de typemap est donné à la fin de cette section.

Si la méthode est définie comme statique, elle appellera la fonction C++ en utilisant la syntaxe classe::methode(). Si la méthode n'est pas statique, la fonction sera appelée avec la syntaxe THIS->method().

Les exemples qui suivent utiliseront la classe C++ que voici :

```
class color {
     public:
     color();
     ~color();
     int blue();
     void set_blue( int );

     private:
     int c_blue;
};
```

Les XSUB pour les méthodes blue() et set_blue() sont définies avec le nom de la classe, mais le paramètre pour l'objet (THIS, ou "self") est implicite et il n'est pas mentionné dans la liste.

```
int
color::blue()

void
color::set_blue( val )
     int val
```

Les deux fonctions attendent un objet en premier paramètre. Le compilateur xsubpp donne à cet objet le nom `THIS`, et l'utilise pour appeler la méthode spécifiée. Ainsi, dans le code C++ généré, les méthodes blue() et set_blue() seront appelées de la façon suivante :

```
RETVAL = THIS->blue();

THIS->set_blue( val );
```

Si le nom de la fonction est **DESTROY**, alors la fonction C++ `delete` sera appelée avec `THIS` comme paramètre.

```
void
color::DESTROY()
```

Le code C++ appellera `delete`.

```
delete THIS;
```

Si le nom de la fonction est **new**, la fonction C++ `new` sera appelée pour créer un objet C++ dynamique. La XSUB s'attendra à recevoir comme premier argument le nom de la classe, qui sera stocké dans une variable nommée `CLASS`.

```
color *
color::new()
```

Le code C++ appellera `new` :

```
   RETVAL = new color();
```

Voici maintenant un exemple de typemap qui pourrait être utilisé dans cet exemple C++.

```
TYPEMAP
color *             O_OBJECT

OUTPUT
# L'objet Perl est beni dans 'CLASS', qui devrait etre un char*
# contenant le nom du paquetage servant a le benir.
O_OBJECT
    sv_setref_pv( $arg, CLASS, (void*)$var );

INPUT
O_OBJECT
    if( sv_isobject($arg) && (SvTYPE(SvRV($arg)) == SVt_PVMG) )
            $var = ($type)SvIV((SV*)SvRV( $arg ));
    else{
            warn( \"${Package}::$func_name() -- $var n'est pas une reference de SV benie\" );
            XSRETURN_UNDEF;
    }
```

2.36 Méthode de construction d'interfaces

Lorsque vous concevez une interface entre Perl et une librairie C, dans de nombreux cas une traduction directe de C vers XS est suffisante. L'interface sera souvent très proche de C et parfois non intuitive, surtout lorsque la fonction C modifie l'un de ses paramètres. Si le programmeur souhaite réaliser une interface de style plus proche de Perl, la méthode suivante peut l'aider à repérer les parties les plus importantes de l'interface.

Repérez les fonctions C qui modifient leurs paramètres. Les XSUB de ces fonctions peuvent retourner à Perl des listes, ou bien, en cas d'échec, undef ou une liste vide.

Repérez les valeurs qui sont utilisées uniquement par les fonctions C et les XSUB. Si le code Perl lui-même n'a pas besoin d'accéder au contenu d'une valeur, alors il n'est peut-être pas nécessaire de fournir une traduction de cette valeur de C en Perl.

Repérez les pointeurs dans la liste de paramètres de la fonction C et dans les valeurs de retour. Certains pointeurs peuvent être traités en XS avec l'opérateur unaire & sur le nom de la variable, tandis que d'autres nécessitent l'utilisation de l'opérateur * sur le nom du type. En général, il est plus facile de travailler avec l'opérateur &.

Repérez les structures utilisées par les fonctions C. On peut souvent utiliser le typemap T_PTROBJ pour faire en sorte que ces structures puissent être manipulées par Perl comme des objets bénis.

2.37 Objets Perl et structures C

Lorsqu'on a affaire à des structures C, il faut choisir soit **T_PTROBJ**, soit **T_PTRREF** pour le type XS. Ces deux types sont conçus pour manipuler des pointeurs vers des objets complexes. Le type T_PTRREF peut être utilisé avec un objet Perl non béni, tandis que le type T_PTROBJ impose que l'objet soit béni. En utilisant T_PTROBJ, on peut obtenir une sorte de vérification de type, car la XSUB essaiera de vérifier que l'objet Perl possède le type attendu.

Le code XS qui suit montre la fonction getnetconfigent() utilisée avec ONC+ TIRPC. Cette fonction retourne un pointeur vers une structure C, et son prototype C est donné plus bas. Cet exemple montrera comment le pointeur C devient une référence Perl. Perl considérera cette référence comme un pointeur sur l'objet béni, et cherchera à appeler un destructeur pour cet objet. Un destructeur sera fourni dans le code XS pour libérer la mémoire utilisée par getnetconfigent(). En XS, les destructeurs peuvent être créés en spécifiant une fonction XSUB dont le nom se termine avec le mot **DESTROY**. Les destructeurs XS peuvent être utilisés pour libérer de la mémoire qui aurait été allouée par malloc() dans une autre XSUB.

```
struct netconfig *getnetconfigent(const char *netid);
```

Un `typedef` peut être créé pour `struct netconfig`. L'objet Perl sera béni dans une classe dont le nom correspondra au type C, avec un suffixe `Ptr` ; le nom ne doit pas contenir de blanc s'il doit devenir un nom de paquetage Perl. Le destructeur de l'objet sera mis dans une classe correspondant à la classe de l'objet, et le mot-clé PREFIX sera utilisé pour réduire le nom au seul mot DESTROY, comme prévu dans Perl.

```
typedef struct netconfig Netconfig;

MODULE = RPC  PACKAGE = RPC

Netconfig *
getnetconfigent(netid)
     char *netid

MODULE = RPC  PACKAGE = NetconfigPtr  PREFIX = rpcb_

void
rpcb_DESTROY(netconf)
     Netconfig *netconf
     CODE:
     printf("Maintenant dans NetconfigPtr::DESTROY\n");
     free( netconf );
```

Cet exemple requiert l'entrée de typemap suivante. Consultez la section sur les typemaps pour plus d'informations sur l'ajout de nouveaux typemaps dans une extension.

```
TYPEMAP
Netconfig *  T_PTROBJ
```

Cet exemple sera utilisé avec l'instruction Perl suivante.

```
use RPC;
$netconf = getnetconfigent("udp");
```

Lorsque Perl détruit l'objet référencé par $netconf, il envoie l'objet à la fonction XSUB DESTROY fournie. Perl ne peut pas déterminer, et ça ne l'intéresse pas, que cet objet est une structure C et non un objet Perl. En ce sens, il n'y a pas de différence entre l'objet créé par la XSUB getnetconfigent() et un objet créé par une sous-routine Perl normale.

2.38 Le typemap

Le typemap est un ensemble de fragments de code utilisés par le compilateur **xsubpp** pour relier les paramètres de fonction et les valeurs C à des valeurs Perl. Le fichier typemap peut contenir trois sections dénommées `TYPEMAP`, `INPUT` et `OUTPUT`. La section `INPUT` indique au compilateur comment convertir des valeurs Perl en valeurs de certains types C. La section OUTPUT indique au compilateur comment traduire les valeurs de certains types C en des valeurs accessibles à Perl. La section TYPEMAP indique au compilateur quels fragments de code, dans les sections INPUT et OUTPUT, doivent être utilisés pour faire correspondre un type C donné à une valeur Perl. Chacune des sections du typemap doit être précédée de l'un des mots-clés TYPEMAP, INPUT et OUTPUT.

Le typemap par défaut, dans le répertoire `ext` du code source Perl, contient un grand nombre de types utiles qui sont mis à disposition des extensions Perl. Certaines extensions définissent des typemaps supplémentaires qu'elles peuvent conserver dans leur propre répertoire. Ces typemaps supplémentaires peuvent se référer aux tables de correspondance INPUT et OUTPUT du typemap principal. Le compilateur **xsubpp** permet au typemap d'une extension de redéfinir des correspondances présentes dans le typemap par défaut.

La plupart des extensions qui nécessitent un typemap personnalisé n'ont besoin que de la section TYPEMAP du fichier typemap. Le typemap personnalisé utilisé dans l'exemple getnetconfigent() présenté précédemment montre ce qui est peut-être l'utilisation typique des typemaps dans les extensions. Ce typemap est utilisé pour faire correspondre une structure C au typemap T_PTROBJ. Le typemap utilisé par getnetconfigent() est présenté ici. Remarquez que le type C est séparé du type XS avec une tabulation, et que l'opérateur unaire `*` de C est considéré comme une partie du nom du type C.

```
TYPEMAP
Netconfig *<tab>T_PTROBJ
```

Voici un exemple plus compliqué. Supposons que vous vouliez que `struct netconfig` soit béni dans la classe `Net::Config`. Une manière de le faire est d'utiliser de la manière suivante le caractère souligné (_) pour séparer les noms de paquetage :

```
typedef struct netconfig * Net_Config;
```

et de fournir ensuite une entrée de typemap `T_PTROBJ_SPECIAL` qui relie le souligné au quadruple point (::), puis de déclarer `Net_Config` avec ce type :

```
TYPEMAP
Net_Config      T_PTROBJ_SPECIAL

INPUT
T_PTROBJ_SPECIAL
        if (sv_derived_from($arg, \"${(my $ntt=$ntype)=~s/_/::/g;\$ntt}\")) {
                IV tmp = SvIV((SV*)SvRV($arg));
        $var = ($type) tmp;
        }
        else
                croak(\"$var n'est pas de type ${(my $ntt=$ntype)=~s/_/::/g;\$ntt}\")

OUTPUT
T_PTROBJ_SPECIAL
        sv_setref_pv($arg, \"${(my $ntt=$ntype)=~s/_/::/g;\$ntt}\",
        (void*)$var);
```

Les sections INPUT et OUTPUT substituent à la volée le souligné au quadruple point, ce qui produit l'effet désiré. Cet exemple donne une idée du pouvoir et de la souplesse du mécanisme des typemaps.

3 EXEMPLES

Fichier `RPC.xs` : interface avec certaines fonctions de la librairie de connexion ONC+ RPC.

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

#include <rpc/rpc.h>

typedef struct netconfig Netconfig;

MODULE = RPC  PACKAGE = RPC

SV *
rpcb_gettime(host="localhost")
     char *host
     PREINIT:
     time_t  timep;
     CODE:
     ST(0) = sv_newmortal();
     if( rpcb_gettime( host, &timep ) )
          sv_setnv( ST(0), (double)timep );

Netconfig *
getnetconfigent(netid="udp")
     char *netid

MODULE = RPC  PACKAGE = NetconfigPtr  PREFIX = rpcb_

void
rpcb_DESTROY(netconf)
     Netconfig *netconf
     CODE:
     printf("NetconfigPtr::DESTROY\n");
     free( netconf );
```

Fichier `typemap` : typemap personnalisé pour RPC.xs.

```
TYPEMAP
Netconfig *  T_PTROBJ
```

Fichier `RPC.pm` : module Perl pour l'extension RPC.

```
package RPC;

require Exporter;
require DynaLoader;
@ISA = qw(Exporter DynaLoader);
@EXPORT = qw(rpcb_gettime getnetconfigent);

bootstrap RPC;
1;
```

Fichier `rpctest.pl` : programme de test Perl pour l'extension RPC.

```
use RPC;

$netconf = getnetconfigent();
$a = rpcb_gettime();
print "time = $a\n";
print "netconf = $netconf\n";

$netconf = getnetconfigent("tcp");
$a = rpcb_gettime("poplar");
print "time = $a\n";
print "netconf = $netconf\n";
```

4 VERSION DE XS

Ce document couvre les fonctionnalités supportées par `xsubpp` 1.935.

5 AUTEUR

Dean Roehrich *<roehrich@cray.com>* Jul 8, 1996.

6 TRADUCTION

6.1 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/.

6.2 Traducteur

Thierry Bézecourt *<thbz@worldnet.fr>*

6.3 Relecture

Régis Julié *<Regis.Julie@cetelem.fr>*

7 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.