NAME

perlxstypemap - Perl XS C/Perl type mapping

DESCRIPTION

The more you think about interfacing between two languages, the more you'll realize that the majority of programmer effort has to go into converting between the data structures that are native to either of the languages involved. This trumps other matter such as differing calling conventions because the problem space is so much greater. There are simply more ways to shove data into memory than there are ways to implement a function call.

Perl XS' attempt at a solution to this is the concept of typemaps. At an abstract level, a Perl XS typemap is nothing but a recipe for converting from a certain Perl data structure to a certain C data structure and vice versa. Since there can be C types that are sufficiently similar to one another to warrant converting with the same logic, XS typemaps are represented by a unique identifier, henceforth called an **XS type** in this document. You can then tell the XS compiler that multiple C types are to be mapped with the same XS typemap.

In your XS code, when you define an argument with a C type or when you are using a `CODE:` and an `OUTPUT:` section together with a C return type of your XSUB, it'll be the typemapping mechanism that makes this easy.

Anatomy of a typemap

In more practical terms, the typemap is a collection of code fragments which are used by the **xsubpp** compiler to map C function parameters and values to Perl values. The typemap file may consist of three sections labelled `TYPEMAP`, `INPUT`, and `OUTPUT`. An unlabelled initial section is assumed to be a `TYPEMAP` section. The INPUT section tells the compiler how to translate Perl values into variables of certain C types. The OUTPUT section tells the compiler how to translate the values from certain C types into values Perl can understand. The TYPEMAP section tells the compiler which of the INPUT and OUTPUT code fragments should be used to map a given C type to a Perl value. The section labels `TYPEMAP`, `INPUT`, or `OUTPUT` must begin in the first column on a line by themselves, and must be in uppercase.

Each type of section can appear an arbitrary number of times and does not have to appear at all. For example, a typemap may commonly lack `INPUT` and `OUTPUT` sections if all it needs to do is associate additional C types with core XS types like T_PTROBJ. Lines that start with a hash# are considered comments and ignored in the `TYPEMAP` section, but are considered significant in `INPUT` and `OUTPUT`. Blank lines are generally ignored.

Traditionally, typemaps needed to be written to a separate file, conventionally called `typemap` in a CPAN distribution. With ExtUtils::ParseXS (the XS compiler) version 3.12 or better which comes with perl 5.16, typemaps can also be embedded directly into XS code using a HERE-doc like syntax:

```
TYPEMAP: <<HERE
...
HERE
```

where `HERE` can be replaced by other identifiers like with normal Perl HERE-docs. All details below about the typemap textual format remain valid.

The `TYPEMAP` section should contain one pair of C type and XS type per line as follows. An example from the core typemap file:

```
TYPEMAP
# all variants of char* is handled by the T_PV typemap
char * T_PV
const char * T_PV
unsigned char * T_PV
...
```

The `INPUT` and `OUTPUT` sections have identical formats, that is, each unindented line starts a new in- or output map respectively. A new in- or output map must start with the name of the XS type to map on a line by itself, followed by the code that implements it indented on the following lines. Example:

```
INPUT
T_PV
$var = ($type)SvPV_nolen($arg)
T_PTR
$var = INT2PTR($type,SvIV($arg))
```

We'll get to the meaning of those Perlish-looking variables in a little bit.

Finally, here's an example of the full typemap file for mapping C strings of the `char *` type to Perl scalars/strings:

```
TYPEMAP
char * T_PV

INPUT
T_PV
$var = ($type)SvPV_nolen($arg)

OUTPUT
T_PV
sv_setpv((SV*)$arg, $var);
```

Here's a more complicated example: suppose that you wanted `struct netconfig` to be blessed into the class `Net::Config` One way to do this is to use underscores (_) to separate package names, as follows:

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

And then provide a typemap entry `T_PTROBJ_SPECIAL` that maps underscores to double-colons (::), and declare `Net_Config` to be of that 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));
$var = INT2PTR($type, tmp);
}
else
croak(\"$var is not of type ${(my $ntt=$ntype)=~s/_/::/g;\$ntt}\")

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

The INPUT and OUTPUT sections substitute underscores for double-colons on the fly, giving the desired effect. This example demonstrates some of the power and versatility of the typemap facility.

The `INT2PTR` macro (defined in perl.h) casts an integer to a pointer of a given type, taking care of the possible different size of integers and pointers. There are also `PTR2IV`, `PTR2UV`, `PTR2NV` macros, to map the other way, which may be useful in OUTPUT sections.

The Role of the typemap File in Your Distribution

The default typemap in the *lib/ExtUtils* directory of the Perl source contains many useful types which can be used by Perl extensions. Some extensions define additional typemaps which they keep in their own directory. These additional typemaps may reference INPUT and OUTPUT maps in the main typemap. The **xsubpp** compiler will allow the extension's own typemap to override any mappings which are in the default

typemap. Instead of using an additional *typemap* file, typemaps may be embedded verbatim in XS with a heredoc-like syntax. See the documentation on the `TYPEMAP:` XS keyword.

For CPAN distributions, you can assume that the XS types defined by the perl core are already available. Additionally, the core typemap has default XS types for a large number of C types. For example, if you simply return a `char *` from your XSUB, the core typemap will have this C type associated with the T_PV XS type. That means your C string will be copied into the PV (pointer value) slot of a new scalar that will be returned from your XSUB to Perl.

If you're developing a CPAN distribution using XS, you may add your own file called *typemap* to the distribution. That file may contain typemaps that either map types that are specific to your code or that override the core typemap file's mappings for common C types.

Sharing typemaps Between CPAN Distributions

Starting with ExtUtils::ParseXS version 3.13_01 (comes with perl 5.16 and better), it is rather easy to share typemap code between multiple CPAN distributions. The general idea is to share it as a module that offers a certain API and have the dependent modules declare that as a built-time requirement and import the typemap into the XS. An example of such a typemap-sharing module on CPAN is `ExtUtils::Typemaps::Basic` Two steps to getting that module's typemaps available in your code:

- Declare `ExtUtils::Typemaps::Basic` as a build-time dependency in `Makefile.PL` (use `BUILD_REQUIRES`), or in your `Build.PL` (use `build_requires`).
- Include the following line in the XS section of your XS file: (don't break the line)

```
INCLUDE_COMMAND: $^X -MExtUtils::Typemaps::Cmd
-e "print embeddable_typemap(q{Basic})"
```

Writing typemap Entries

Each INPUT or OUTPUT typemap entry is a double-quoted Perl string that will be evaluated in the presence of certain variables to get the final C code for mapping a certain C type.

This means that you can embed Perl code in your typemap (C) code using constructs such as `${ perl code that evaluates to scalar reference here }`. A common use case is to generate error messages that refer to the true function name even when using the ALIAS XS feature:

```
${ $ALIAS ? \q[GvNAME(CvGV(cv))] : \qq[\"$pname\"] }
```

For many typemap examples, refer to the core typemap file that can be found in the perl source tree at *lib/ExtUtils/typemap*.

The Perl variables that are available for interpolation into typemaps are the following:

- *`$var`* - the name of the input or output variable, eg. RETVAL for return values.
- *`$type`* - the raw C type of the parameter, any `:` replaced with `_`. e.g. for a type of`Foo::Bar` *`$type`* is `Foo__Bar`
- *`$ntype`* - the supplied type with `*` replaced with `Ptr`. e.g. for a type of`Foo*`, *`$ntype`* is `FooPtr`
- *`$arg`* - the stack entry, that the parameter is input from or output to, e.g. `ST(0)`
- *`$argoff`* - the argument stack offset of the argument. ie. 0 for the first argument, etc.
- *`$pname`* - the full name of the XSUB, with including the `PACKAGE` name, with any `PREFIX` stripped. This is the non-ALIAS name.
- *`$Package`* - the package specified by the most recent `PACKAGE` keyword.
- *`$ALIAS`* - non-zero if the current XSUB has any aliases declared with `ALIAS`.

Full Listing of Core Typemaps

Each C type is represented by an entry in the typemap file that is responsible for converting perl variables (SV, AV, HV, CV, etc.) to and from that type. The following sections list all XS types that come with perl by default.

T_SV
: This simply passes the C representation of the Perl variable (an SV*) in and out of the XS layer. This can be used if the C code wants to deal directly with the Perl variable.

T_SVREF
: Used to pass in and return a reference to an SV.

  Note that this typemap does not decrement the reference count when returning the reference to an SV*. See also: T_SVREF_REFCOUNT_FIXED

T_SVREF_FIXED
: Used to pass in and return a reference to an SV. This is a fixed variant of T_SVREF that decrements the refcount appropriately when returning a reference to an SV*. Introduced in perl 5.15.4.

T_AVREF
: From the perl level this is a reference to a perl array. From the C level this is a pointer to an AV.

  Note that this typemap does not decrement the reference count when returning an AV*. See also: T_AVREF_REFCOUNT_FIXED

T_AVREF_REFCOUNT_FIXED
: From the perl level this is a reference to a perl array. From the C level this is a pointer to an AV. This is a fixed variant of T_AVREF that decrements the refcount appropriately when returning an AV*. Introduced in perl 5.15.4.

T_HVREF
: From the perl level this is a reference to a perl hash. From the C level this is a pointer to an HV.

  Note that this typemap does not decrement the reference count when returning an HV*. See also: T_HVREF_REFCOUNT_FIXED

T_HVREF_REFCOUNT_FIXED
: From the perl level this is a reference to a perl hash. From the C level this is a pointer to an HV. This is a fixed variant of T_HVREF that decrements the refcount appropriately when returning an HV*. Introduced in perl 5.15.4.

T_CVREF
: From the perl level this is a reference to a perl subroutine (e.g. `$sub` = sub { 1 };). From the C level this is a pointer to a CV.

  Note that this typemap does not decrement the reference count when returning an HV*. See also: T_HVREF_REFCOUNT_FIXED

T_CVREF_REFCOUNT_FIXED
: From the perl level this is a reference to a perl subroutine (e.g. `$sub` = sub { 1 };). From the C level this is a pointer to a CV.

  This is a fixed variant of T_HVREF that decrements the refcount appropriately when returning an HV*. Introduced in perl 5.15.4.

T_SYSRET
: The T_SYSRET typemap is used to process return values from system calls. It is only meaningful when passing values from C to perl (there is no concept of passing a system return value from Perl to C).

  System calls return -1 on error (setting ERRNO with the reason) and (usually) 0 on success. If the return value is -1 this typemap returns `undef`. If the return value is not -1, this typemap translates a 0 (perl false) to ‘‘0 but true’’ (which is perl true) or returns the value itself, to indicate that the command succeeded.

  The POSIX module makes extensive use of this type.

T_UV
: An unsigned integer.

T_IV
: A signed integer. This is cast to the required integer type when passed to C and converted to an IV when passed back to Perl.

T_INT
: A signed integer. This typemap converts the Perl value to a native integer type (the `int` type on the current platform). When returning the value to perl it is processed in the same way as for T_IV.

  Its behaviour is identical to using an `int` type in XS with T_IV.

T_ENUM
: An enum value. Used to transfer an enum component from C. There is no reason to pass an enum value to C since it is stored as an IV inside perl.

T_BOOL
: A boolean type. This can be used to pass true and false values to and from C.

T_U_INT
: This is for unsigned integers. It is equivalent to using T_UV but explicitly casts the variable to type `unsigned int`. The default type for `unsigned int` is T_UV.

T_SHORT
: Short integers. This is equivalent to T_IV but explicitly casts the return to type `short`. The default typemap for `short` is T_IV.

T_U_SHORT
: Unsigned short integers. This is equivalent to T_UV but explicitly casts the return to type `unsigned short`. The default typemap for `unsigned short` is T_UV.

  T_U_SHORT is used for type `U16` in the standard typemap.

T_LONG
: Long integers. This is equivalent to T_IV but explicitly casts the return to type `long`. The default typemap for `long` is T_IV.

T_U_LONG
: Unsigned long integers. This is equivalent to T_UV but explicitly casts the return to type `unsigned long`. The default typemap for `unsigned long` is T_UV.

  T_U_LONG is used for type `U32` in the standard typemap.

T_CHAR
: Single 8-bit characters.

T_U_CHAR
: An unsigned byte.

T_FLOAT
: A floating point number. This typemap guarantees to return a variable cast to a `float`.

T_NV
: A Perl floating point number. Similar to T_IV and T_UV in that the return type is cast to the requested numeric type rather than to a specific type.

T_DOUBLE
: A double precision floating point number. This typemap guarantees to return a variable cast to a `double`.

T_PV
: A string (char *).

T_PTR
: A memory address (pointer). Typically associated with a `void *` type.

T_PTRREF
: Similar to T_PTR except that the pointer is stored in a scalar and the reference to that scalar is returned to the caller. This can be used to hide the actual pointer value from the programmer since it is usually not required directly from within perl.

  The typemap checks that a scalar reference is passed from perl to XS.

T_PTROBJ
: Similar to T_PTRREF except that the reference is blessed into a class. This allows the pointer to be used as an object. Most commonly used to deal with C structs. The typemap checks that the perl object passed into the XS routine is of the correct class (or part of a subclass).

  The pointer is blessed into a class that is derived from the name of type of the pointer but with all '*' in the name replaced with 'Ptr'.

  For `DESTROY` XSUBs only, a T_PTROBJ is optimized to a T_PTRREF. This means the class check is skipped.

T_REF_IV_REF
: NOT YET

T_REF_IV_PTR
: Similar to T_PTROBJ in that the pointer is blessed into a scalar object. The difference is that when the object is passed back into XS it must be of the correct type (inheritance is not supported) while T_PTROBJ supports inheritance.

  The pointer is blessed into a class that is derived from the name of type of the pointer but with all '*' in the name replaced with 'Ptr'.

  For `DESTROY` XSUBs only, a T_REF_IV_PTR is optimized to a T_PTRREF. This means the class check is skipped.

T_PTRDESC
: NOT YET

T_REFREF
: Similar to T_PTRREF, except the pointer stored in the referenced scalar is dereferenced and copied to the output variable. This means that T_REFREF is to T_PTRREF as T_OPAQUE is to T_OPAQUEPTR. All clear?

  Only the INPUT part of this is implemented (Perl to XSUB) and there are no known users in core or on CPAN.

T_REFOBJ
: Like T_REFREF, except it does strict type checking (inheritance is not supported).

  For `DESTROY` XSUBs only, a T_REFOBJ is optimized to a T_REFREF. This means the class check is skipped.

T_OPAQUEPTR
: This can be used to store bytes in the string component of the SV. Here the representation of the data is irrelevant to perl and the bytes themselves are just stored in the SV. It is assumed that the C variable is a pointer (the bytes are copied from that memory location). If the pointer is pointing to something that is represented by 8 bytes then those 8 bytes are stored in the SV (and *length()* will report a value of 8). This entry is similar to T_OPAQUE.

  In principle the *unpack()* command can be used to convert the bytes back to a number (if the underlying type is known to be a number).

  This entry can be used to store a C structure (the number of bytes to be copied is calculated using the C `sizeof` function) and can be used as an alternative to T_PTRREF without having to worry about a

memory leak (since Perl will clean up the SV).

T_OPAQUE

This can be used to store data from non-pointer types in the string part of an SV. It is similar to T_OPAQUEPTR except that the typemap retrieves the pointer directly rather than assuming it is being supplied. For example, if an integer is imported into Perl using T_OPAQUE rather than T_IV the underlying bytes representing the integer will be stored in the SV but the actual integer value will not be available. i.e. The data is opaque to perl.

The data may be retrieved using the `unpack` function if the underlying type of the byte stream is known.

T_OPAQUE supports input and output of simple types. T_OPAQUEPTR can be used to pass these bytes back into C if a pointer is acceptable.

Implicit array

xsubpp supports a special syntax for returning packed C arrays to perl. If the XS return type is given as

```
array(type, nelem)
```

xsubpp will copy the contents of `nelem * sizeof(type)` bytes from RETVAL to an SV and push it onto the stack. This is only really useful if the number of items to be returned is known at compile time and you don't mind having a string of bytes in your SV. Use T_ARRAY to push a variable number of arguments onto the return stack (they won't be packed as a single string though).

This is similar to using T_OPAQUEPTR but can be used to process more than one element.

T_PACKED

Calls user-supplied functions for conversion. For `OUTPUT` (XSUB to Perl), a function named `XS_pack_$ntype` is called with the output Perl scalar and the C variable to convert from. `$ntype` is the normalized C type that is to be mapped to Perl. Normalized means that all `*` are replaced by the string `Ptr`. The return value of the function is ignored.

Conversely for `INPUT` (Perl to XSUB) mapping, the function named `XS_unpack_$ntype` is called with the input Perl scalar as argument and the return value is cast to the mapped C type and assigned to the output C variable.

An example conversion function for a typemapped struct `foo_t *` might be:

```
static void
XS_pack_foo_tPtr(SV *out, foo_t *in)
{
dTHX; /* alas, signature does not include pTHX_ */
HV* hash = newHV();
hv_stores(hash, "int_member", newSViv(in->int_member));
hv_stores(hash, "float_member", newSVnv(in->float_member));
/* ... */

/* mortalize as thy stack is not refcounted */
sv_setsv(out, sv_2mortal(newRV_noinc((SV*)hash)));
}
```

The conversion from Perl to C is left as an exercise to the reader, but the prototype would be:

```
static foo_t *
XS_unpack_foo_tPtr(SV *in);
```

Instead of an actual C function that has to fetch the thread context using `dTHX`, you can define macros of the same name and avoid the overhead. Also, keep in mind to possibly free the memory allocated by `XS_unpack_foo_tPtr`.

T_PACKEDARRAY

T_PACKEDARRAY is similar to T_PACKED. In fact, the `INPUT` (Perl to XSUB) typemap is identical, but the `OUTPUT` typemap passes an additional argument to the `XS_pack_$ntype` function. This third parameter indicates the number of elements in the output so that the function can handle C arrays sanely. The variable needs to be declared by the user and must have the name `count_$ntype` where `$ntype` is the normalized C type name as explained above. The signature of the function would be for the example above and `foo_t **`:

```
static void
XS_pack_foo_tPtrPtr(SV *out, foo_t *in, UV count_foo_tPtrPtr);
```

The type of the third parameter is arbitrary as far as the typemap is concerned. It just has to be in line with the declared variable.

Of course, unless you know the number of elements in the `sometype **` C array, within your XSUB, the return value from `foo_t ** XS_unpack_foo_tPtrPtr(...)` will be hard to decipher. Since the details are all up to the XS author (the typemap user), there are several solutions, none of which particularly elegant. The most commonly seen solution has been to allocate memory for N+1 pointers and assign `NULL` to the (N+1)th to facilitate iteration.

Alternatively, using a customized typemap for your purposes in the first place is probably preferable.

T_DATAUNIT

NOT YET

T_CALLBACK

NOT YET

T_ARRAY

This is used to convert the perl argument list to a C array and for pushing the contents of a C array onto the perl argument stack.

The usual calling signature is

```
@out = array_func( @in );
```

Any number of arguments can occur in the list before the array but the input and output arrays must be the last elements in the list.

When used to pass a perl list to C the XS writer must provide a function (named after the array type but with 'Ptr' substituted for '*') to allocate the memory required to hold the list. A pointer should be returned. It is up to the XS writer to free the memory on exit from the function. The variable `ix_$var` is set to the number of elements in the new array.

When returning a C array to Perl the XS writer must provide an integer variable called `size_$var` containing the number of elements in the array. This is used to determine how many elements should be pushed onto the return argument stack. This is not required on input since Perl knows how many arguments are on the stack when the routine is called. Ordinarily this variable would be called `size_RETVAL`.

Additionally, the type of each element is determined from the type of the array. If the array uses type `intArray *`xsubpp will automatically w ork out that it contains variables of type `int` and use that typemap entry to perform the copy of each element. All pointer '*' and 'Array' tags are removed from the name to determine the subtype.

T_STDIO

This is used for passing perl filehandles to and from C using `FILE *` structures.

T_INOUT

This is used for passing perl filehandles to and from C using `PerlIO *` structures. The file handle can used for reading and writing. This corresponds to the `+<` mode, see also T_IN and T_OUT.

See perliol(1) for more information on the Perl IO abstraction layer. Perl must have been built with

`-Duseperlio`.

There is no check to assert that the filehandle passed from Perl to C was created with the right `open()` mode.

Hint: The perlxstut(1) tutorial covers the T_INOUT, T_IN, and T_OUT XS types nicely.

T_IN
: Same as T_INOUT, but the filehandle that is returned from C to Perl can only be used for reading (mode <).

T_OUT
: Same as T_INOUT, but the filehandle that is returned from C to Perl is set to use the open mode +>.