NAME

ExtUtils::XSpp − XS for C++

SYNOPSIS

```
xspp [--typemap=typemap.xsp [--typemap=typemap2.xsp]]
     [--xsubpp[=/path/to/xsubpp] [--xsubpp-args="xsubpp args"]
     Foo.xsp
```

or

```
perl -MExtUtils::XSpp::Cmd -e xspp -- <xspp options and arguments>
```

In Foo.xs

```
INCLUDE_COMMAND: $^X -MExtUtils::XSpp::Cmd -e xspp -- <xspp options/arguments>
```

Using `ExtUtils::XSpp::Cmd` is equivalent to using the `xspp` command line script, except that there is no guarantee for `xspp` to be installed in the system PATH.

OVERVIEW

XS++ is just a thin layer over plain XS, hence to use it you are supposed to know, at the very least, C++ and XS.

This means that you may need typemaps for **both** the normal XS pre-processor *xsubpp* and the XS++ pre-processor *xspp*. More on that in the *TYPEMAPS* section below.

COMMAND LINE

`--typemap=/path/to/typemap.xsp`

Can be specified multiple times to process additional typemap files before the main XS++ input files. Typemap files are processed the same way as regular XS++ files, except that output code is discarded.

`--xsubpp[=/path/to/xsubpp]`

If specified, XS++ will run *xsubpp* after processing the XS++ input file. If the path to *xsubpp* is not specified, *xspp* expects to find it in the system PATH.

`--xsubpp-args=''extra xsubpp args''`

Can be used to pass additional command line arguments to *xsubpp*.

TYPEMAPS

Ordinary XS typemaps

To recap, *ordinary* XS typemaps do the following three things:

- Associate a C type with an identifier such as T_FOO or O_FOO (which we'll call *XS type* here).
- Define an INPUT mapping for converting a Perl data structure to the aforementioned C type.
- Define an OUTPUT mapping for converting the C data structure back into a Perl data structure.

These are still required in the context of XS++. There are some helpers to take away the tedium, but I'll get to that later. For XS++, there's another layer of typemaps. The following section will discuss those.

XS++ typemaps

There is nothing special about XS++ typemap files (i.e. you can put typemaps directly in your *.xsp* file), but it is handy to have common typemaps in a separate file, typically called *typemap.xsp* to avoid duplication.

```
%typemap{<C++ type>}{simple};
```

Just let XS++ know that this is a valid type, the type will be passed unchanged to XS code **except** that any `const` qualifiers will be stripped.

```
%typemap{<C++ reference type>}{reference};
```

Handle C++ references: the XS variable will be declared as a pointer, and it will be explicitly dereferenced in the function call. If it is used in the return value, the function will create **copy** of the returned value using a copy constructor.

As a shortcut for the common case of declaring both of the above for a given type, you may use

```
%typemap{<C++ type>};
```

Which has the same effect as:

```
%typemap{<C++ type>}{simple};
%typemap{<C++ type>&}{reference};
```

For more control over the type mapping, you can use the `parsed` variant as follows.

```
%typemap{<C++ type 1>}{parsed}{%<C++ type 2>%};
```

When `C++ type 1` is used, replace it with `C++ type 2` in the generated XS code.

```
%typemap{<C++ type>}{parsed}{
    %cpp_type{%<C++ type 2>%};
    %call_function_code{% $CVar = new Foo( $Call ) %};
    %cleanup_code{% ... %};
    %precall_code{% ... %};

    # use only one of the following
    %output_code{% $PerlVar = newSViv( $CVar ) %};
    %output_list{% PUTBACK; XPUSHi( $CVar ); SPAGAIN %};
};
```

Is a more flexible form for the `parsed` typemap. All the parameters are optional.

cpp_type
: Specifies the C++ type used for the variable declaration in the generated XS code.

  If not specified defaults to the type specified in the typemap.

call_function_code
: Used when the typemap applies to the return value of the function.

  Specifies the code to use in the function call. The special variables `$Call` and `$CVar` are replaced with the actual call code and the name of the C++ return variable.

output_code
: Used when the typemap applies to the return value of the function. See also `%output_list`.

  Specifies the code emitted right after the function call to convert the C++ return value into a Perl return value. The special variable `$CVar` is replaced with the C++ return variable name.

cleanup_code
: Used when the typemap applies to the return value of the function.

  Specifies some code emitted after output value processing. The special variables `$PerlVar` and `$CVar` are replaced with the names of the C++ variables containing the Perl scalar and the corresponding C++ value.

precall_code
: Used when the typemap applies to a parameter.

  Specifies some code emitted after argument processing and before calling the C++ method. The special variables `$PerlVar` and `$CVar` are replaced with the names of the C++ variables containing the Perl scalar and the corresponding C++ value.

output_list
: Used when the typemap applies to the return value of the function, as an alternative to `%output_code`.

  Specifies some code that manipulates the Perl stack directly in order to return a list. The special variable `$CVar` is replaced with the C++ name of the output variable.

  The code must use PUTBACK/SPAGAIN if appropriate.

Putting all the typemaps together

In summary, the XS++ typemaps (optionally) give you much more control over the type conversion code that's generated for your XSUBs. But you still need to let the XS compiler know how to map the C types to Perl and back using the XS typemaps.

Most of the time, you just need to convert basic C(++) types or the types that you define with your C++ classes. For the former, XS++ comes with a few default mappings for booleans, integers, floating point numbers, and strings. For classes, XS++ can automatically create a mapping of type `O_OBJECT` which

uses the de-facto standard way of storing a pointer to the C(++) object in the IV slot of a referenced/blessed scalar. Due to backwards compatibility, this must be explicitly enabled by adding

```
%loadplugin{feature::default_xs_typemap};
```

in *typemap.xsp* (or near the top of every *.xsp* file).

If you deal with any other types as arguments or return types, you still need to write both XS and XS++ typemaps for these so that the systems know how to deal with them.

See either ''Custom XS typemaps'' below for a way to specify XS typemaps from XS++ or perlxs for a discussion of inline XS typemaps that don't require the traditional XS *typemap* file.

Custom XS typemaps

XS++ provides a default mapping for object types to an `O_OBJECT` typemap with standard input and output glue code, which should be adequate for most uses.

There are multiple ways to override this default when needed.

```
%typemap{Foo *}{simple}{
    %xs_type{O_MYMAP};
    %xs_input_code{% ... %}; // optional
    %xs_output_code{% ... %}; // optional
};
```

can be used to define a new type -> XS typemap mapping, with optinal input/output code. Since XS typemap definitions are global, XS input/output code applies to all types with the same `%xs_type`, hence there is no need to repeat it.

```
%typemap{_}{simple}{
    %name{object};
    %xs_type{O_MYMAP};
    %xs_input_code{% ... %}; // optional
    %xs_output_code{% ... %}; // optional
};
```

can be used to change the default typemap used for all classes.

DESCRIPTION

Anything that does not look like a XS++ directive or a class declaration is passed verbatim to XS. If you want XS++ to ignore code that looks like a XS++ directive or class declaration, simply surround it with a raw block delimiter like this:

```
%{
XS++ won't interpret this
%}
```

`%code`

See under **Classes**. Note that custom `%code` blocks are the only exception to the exception handling. By specifying a custom `%code` block, you forgo the automatic exception handlers.

`%file`

```
%file{file/path.h};
...
%file{file/path2};
...
%file{-}
```

By default XS++ output goes to standard output; to change this, use the `%file` directive; use `-` for standard output.

`%module`

```
%module{Module::Name};
```

Will be used to generate the `MODULE=Module::Name` XS directives. It indirectly sets the name of the shared library that is generated as well as the name of the module via which XSLoader will be able to find/load it.

`%name`

```
%name{Perl::Class} class MyClass { ... };
%name{Perl::Func} int foo();
```

Specifies the Perl name under which the C++ class/function will be accessible. By default, constructor names are mapped to `new` in Perl.

`%typemap`

See "TYPEMAPS" above.

`%length`

When you need to pass a string from Perl to an XSUB that takes the C string and its length as arguments, you may have XS++ pass the length of the string automatically. For example, if you declare a method as follows,

```
void PrintLine( char* line, unsigned int %length{line} );
```

you can call the method from Perl like this:

```
$object->PrintLine( $string );
```

This feature is also present in plain XS. See also: perlxs.

If you use `%length(line)` in conjunction with any kind of special code block such as `%code`, `%postcall`, etc., then you can refer to the length of the string (here: `line`) *efficiently* as `length(line)` in the code.

`%alias`

Decorator for function/method declarations such as

```
double add(double a, double b)
  %alias{subtract = 1} %alias{multiply = 2};
```

Which will cause the generation of just a single XSUB using the XS "ALIAS" feature (see perlxs). It will be installed as all of `add`, `subtract`, and `multiply` on the Perl side and call either the C++ `add`, `subtract`, or `multiply` functions depending on which way it was called.

Notes: If used in conjunction with `%name{foo}` to rename the function, then the `%name` will only affect the main function name (in the above example, `add` but not `subtract` or `multiply`). When used with the `%code` feature, the custom code will have to use the `ix` integer variable to decide which function to call. `ix` is set to 0 for the main function. Make sure to read up on the ALIAS feature of plain XS. Aliasing is not supported for constructors and destructors.

Classes

```
%name{My::Class} class MyClass : public %name{My::Base} MyBase
{
    // can be called in Perl as My::Class->new( ... );
    MyClass( int arg );
    // My::Class->newMyClass( ... );
    %name{newMyClass} MyClass( const char* str, int arg );

    // standard DESTROY method
    ~MyClass();

    int GetInt();
    void SetValue( int arg = -1 );

    %name{SetString} void SetValue( const char* string = NULL );

    // Supply a C<CODE:> or C<CLEANUP:> block for the XS
    int MyMethod( int a, int b )
        %code{% RETVAL = a + b; %}
        %cleanup{% /* do something */ %};

    // Expose class method as My::ClassMethod::ClassMethod($foo)
    static void ClassMethod( double foo );
```

```
        // Expose member variable as a pair of set_boolean/get_boolean accessors
        bool boolean %get %set;
};
```

Comments

XS++ recognizes both C−style comments `/* ... */` and C++−style comments `// ...`. Comments are removed from the XS output.

Exceptions

C++ Exceptions are always caught and transformed to Perl `croak()` calls. If the exception that was caught inherited from `std::exception`, then the `what()` message is included in the Perl-level error message. All other exceptions will result in the `croak()` message `"Caught unhandled C++ exception of unknown type"`.

Note that if you supply a custom `%code` block for a function or method, the automatic exception handling is turned off.

Member variables

By default, member variable declarations are ignored; the `%get` and `%set` decorators syntehsize a getter/setter named after the member variable (can be renamed using `%name`).

XS++ defaults to get_/set_ prefix for getters/setters. This can be overridden on an individual basis by using e.g.

```
int foo %get{readFoo} %set{writeFoo};
```

As an alternative, the class-level `%accessors` decorator sets the the accessor style for the whole class:

```
%accessors{
    %get_style{no_prefix};
    %set_style{camelcase};
};
```

Available styles are

no_prefix => foo
underscore => get_foo, set_foo
camelcase => getFoo, setFoo
uppercase => GetFoo, SetFoo

EXAMPLES

The distribution contains an *examples* directory. The *examples/XSpp−Example* directory therein demonstrates a particularly simple way of getting started with XS++.

AUTHOR

Mattia Barbon <mbarbon AT cpan DOT org>

LICENSE