NAME

Module::Build::WithXSpp − XS++ enhanced flavour of Module::Build

SYNOPSIS

In *Build.PL*:

```
use strict;
use warnings;
use 5.006001;

use Module::Build::WithXSpp;

my $build = Module::Build::WithXSpp−>new(
  # normal Module::Build arguments...
  # optional: mix in some extra C typemaps:
  extra_typemap_modules => {
    'ExtUtils::Typemaps::ObjectMap' => '0',
  },
);
$build−>create_build_script;
```

DESCRIPTION

This subclass of Module::Build adds some tools and processes to make it easier to use for wrapping C++ using XS++ (ExtUtils::XSpp).

There are a few minor differences from using `Module::Build` for an ordinary XS module and a few conventions that you should be aware of as an XS++ module author. They are documented in the ‘‘FEATURES AND CONVENTIONS’’ section below. But if you can’t be bothered to read all that, you may choose skip it and blindly follow the advice in ‘‘JUMP START FOR THE IMPATIENT’’.

An example of a full distribution based on this build tool can be found in the ExtUtils::XSpp distribution under *examples/XSpp−Example*. Using that example as the basis for your `Module::Build::WithXSpp`−based distribution is probably a good idea.

FEATURES AND CONVENTIONS

XS files

By default, `Module::Build::WithXSpp` will automatically generate a main XS file for your module which includes all XS++ files and does the correct incantations to support C++.

If `Module::Build::WithXSpp` detects any XS files in your module, it will skip the generation of this default file and assume that you wrote a custom main XS file. If that is not what you want, and wish to simply include plain XS code, then you should put the XS in a verbatim block of an *.xsp* file. In case you need to use the plain-C part of an XS file for `#include` directives and other code, then put your code into a header file and `#include` it from an *.xsp* file:

In *src/mystuff.h*:

```
#include <something>
using namespace some::thing;
```

In *xsp/MyClass.xsp*

```
#include "mystuff.h"

%{
  ... verbatim XS here ...
%}
```

Note that there is no guarantee about the order in which the XS++ files are picked up.

Build directory

When building your XS++ based extension, a temporary build directory *buildtmp* is created for the byproducts. It is automatically cleaned up by `./Build clean`.

Source directories

A Perl module distribution typically has the module `.pm` files in its *lib* subdirectory. In a `Module::Build::WithXSpp` based distribution, there are two more such conventions about source directories:

If any C++ source files are present in the *src* directory, they will be compiled to object files and linked automatically.

Any `.xs`, `.xsp`, and `.xspt` files in an *xs* or *xsp* subdirectory will be automatically picked up and included by the build system.

For backwards compatibility, files of the above types are also recognized in *lib*.

Typemaps

In XS++, there are two types of typemaps: The ordinary XS typemaps which conventionally put in a file called *typemap*, and XS++ typemaps.

The ordinary XS typemaps will be found in the main directory, under *lib*, and in the XS directories (*xs* and *xsp*). They are required to carry the `.map` extension or to be called *typemap*. You may use multiple *.map* files if the entries do not collide. They will be merged at build time into a complete *typemap* file in the temporary build directory.

The `extra_typemap_modules` option is the preferred way to do XS typemapping. It works like any other `Module::Build` argument that declares dependencies except that it loads the listed modules at build time and includes their typemaps into the build.

The XS++ typemaps are required to carry the `.xspt` extension or (for backwards compatibility) to be called `typemap.xsp`.

Detecting the C++ compiler

`Module::Build::WithXSpp` uses ExtUtils::CppGuess to detect a C++ compiler on your system that is compatible with the C compiler that was used to compile your perl binary. It sets some additional compiler/linker options.

This is known to work on GCC (Linux, MacOS, Windows, and ?) as well as the MS VC toolchain. Patches to enable other compilers are **very** welcome.

Automatic dependencies

`Module::Build::WithXSpp` automatically adds several dependencies (on the currently running versions) to your distribution. You can disable this by setting `auto_configure_requires => 0` in *Build.PL*.

These are at configure time: `Module::Build`, `Module::Build::WithXSpp` itself, and `ExtUtils::CppGuess`. Additionally there will be a build-time dependency on `ExtUtils::XSpp`.

You do not have to set these dependencies yourself unless you need to set the required versions manually.

Include files

Unfortunately, including the perl headers produces quite some pollution and redefinition of common symbols. Therefore, it may be necessary to include some of your headers before including the perl headers. Specifically, this is the case for MSVC compilers and the standard library headers.

Therefore, if you care about that platform in the least, you should use the `early_includes` option when creating a `Module::Build::WithXSpp` object to list headers to include before the perl headers. If such a supplied header file starts with a double quote, `#include "..."` is used, otherwise `#include <...>` is the default. Example:

```
Module::Build::WithXSpp->new(
  early_includes => [qw(
    "mylocalheader.h"
    <mysystemheader.h>
  )]
)
```

JUMP START FOR THE IMPATIENT

There are as many ways to start a new CPAN distribution as there are CPAN distributions. Choose your favourite (I just do `h2xs -An My::Module`), then apply a few changes to your setup:

- Obliterate any *Makefile.PL*.

This is what your *Build.PL* should look like:

```
use strict;
use warnings;
use 5.006001;
use Module::Build::WithXSpp;

my $build = Module::Build::WithXSpp->new(
  module_name         => 'My::Module',
  license             => 'perl',
  dist_author         => q{John Doe <john_does_mail_address>},
  dist_version_from   => 'lib/My/Module.pm',
  build_requires => { 'Test::More' => 0, },
  extra_typemap_modules => {
    'ExtUtils::Typemaps::ObjectMap' => '0',
    # ...
  },
);
$build->create_build_script;
```

If you need to link against some library `libfoo`, add this to the options:

```
  extra_linker_flags => [qw(-lfoo)],
```

There is `extra_compiler_flags`, too, if you need it.

- You create two folders in the main distribution folder: *src* and *xsp*.
- You put any C++ code that you want to build and include in the module into *src/*. All the typical C(++) file extensions are recognized and will be compiled to object files and linked into the module. And headers in that folder will be accessible for `#include <myheader.h>`.

  For good measure, move a copy of *ppport.h* to that directory. See Devel::PPPort.
- You do not write normal XS files. Instead, you write XS++ and put it into the *xsp/* folder in files with the `.xsp` extension. Do not worry, you can include verbatim XS blocks in XS++. For details on XS++, see ExtUtils::XSpp.
- If you need to do any XS type mapping, put your typemaps into a *.map* file in the `xsp` directory. Alternatively, search CPAN for an appropriate typemap module (cf. ExtUtils::Typemaps::Default for an explanation). XS++ typemaps belong into *.xspt* files in the same directory.
- In this scheme, *lib/* only contains Perl module files (and POD). If you started from a pure-Perl distribution, don't forget to add these magic two lines to your main module:

  ```
  require XSLoader;
  XSLoader::load('My::Module', $VERSION);
  ```

SEE ALSO

Module::Build upon which this module is based.

ExtUtils::XSpp implements XS++. The `ExtUtils::XSpp` distribution contains an *examples* directory with a usage example of this module.

ExtUtils::Typemaps implements progammatic modification (merging) of C/XS typemaps. `ExtUtils::Typemaps` was renamed from `ExtUtils::Typemap` since the original name conflicted with the core *typemap* file on case-insensitive file systems.

ExtUtils::Typemaps::Default explains the concept of having typemaps shipped as modules.

ExtUtils::Typemaps::ObjectMap is such a typemap module and probably very useful for any XS++ module.

ExtUtils::Typemaps::STL::String implements simple typemapping for STL `std::string`s.

AUTHOR

Steffen Mueller <smueller AT cpan DOT org>

With input and bug fixes from:

Mattia Barbon

Shmuel Fomberg

Florian Schlichting

COPYRIGHT AND LICENSE