Baby XS: Just enough to get you started
YAPC::NA 2012

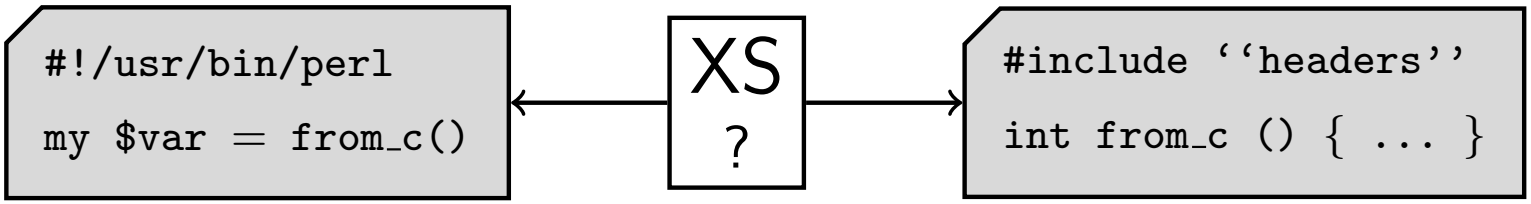
Joel Berger

University of Illinois at Chicago

June 15, 2012

What is XS?

- XS the mechanism used to connect Perl and C
  - write optimized functions
  - access C libraries

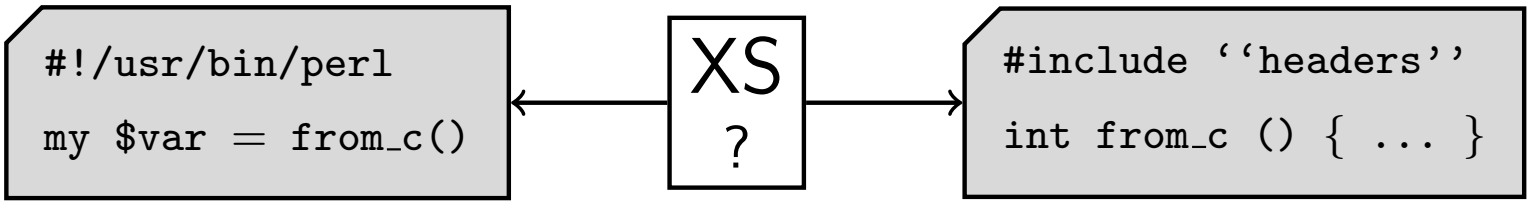


But what *is* XS?

- XS is C functions/macros provided by Perl headers
- XS is also XS-preprocessesor directives

What is XS?

- XS the mechanism used to connect Perl and C
  - write optimized functions
  - access C libraries

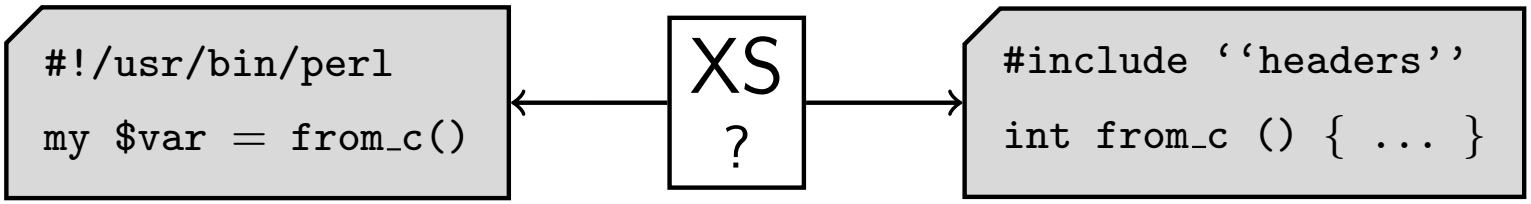


But what *is* XS?

- XS is C functions/macros provided by Perl headers
- XS is also XS-preprocessesor directives

What is XS?

- XS the mechanism used to connect Perl and C
  - write optimized functions
  - access C libraries

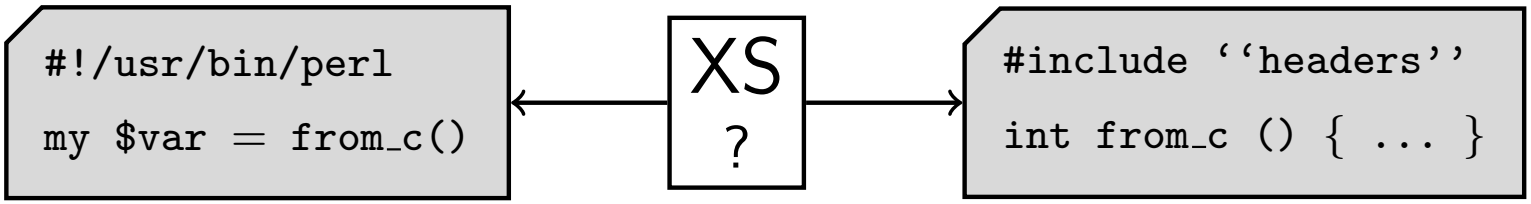


But what *is* XS?

- XS is C functions/macros provided by Perl headers
- XS is also XS-preprocessesor directives

What is XS?

- XS the mechanism used to connect Perl and C
  - write optimized functions
  - access C libraries



But what *is* XS?

- XS is C functions/macros provided by Perl headers
- XS is also XS-preprocessesor directives

What are "baby" languages?

"Baby" Langauge

The naive code that new programmers write,
a simple subset of the full language

- "Baby Perl"
  - no use of `map` / `grep`
  - avoids references
  - avoids `$_` and other special variables
- "Full XS" is powerful but is lots to learn
- "Baby XS"
  - looks like C
  - behaves like Perl

What are "baby" languages?

"Baby" Langauge

The naive code that new programmers write,
a simple subset of the full language

- "Baby Perl"
  - no use of `map` / `grep`
  - avoids references
  - avoids `$_` and other special variables
- "Full XS" is powerful but is lots to learn
- "Baby XS"
  - looks like C
  - behaves like Perl

What are "baby" languages?

"Baby" Langauge

The naive code that new programmers write,
a simple subset of the full language

- "Baby Perl"
  - no use of `map` / `grep`
  - avoids references
  - avoids `$_` and other special variables
- "Full XS" is powerful but is lots to learn
- "Baby XS"
  - looks like C
  - behaves like Perl

What are "baby" languages?

"Baby" Langauge

The naive code that new programmers write,
a simple subset of the full language

- "Baby Perl"
  - no use of `map` / `grep`
  - avoids references
  - avoids `$_` and other special variables
- "Full XS" is powerful but is lots to learn
- "Baby XS"
  - looks like C
  - behaves like Perl

Baby XS

What is Baby XS?

Some "easy" idioms and rules-of-thumb to keep XS from becoming overwhelming

- ignores typemaps
- uses Perl datatype-specific functions from `perldoc perlguts`
- ignores most of the XSpp commands
- uses a Perl-level function wrapper to munge input/output if needed
  - ignores stack macros
  - avoids (most) issues of "mortalization"
- expands to "real" XS

Baby XS

What is Baby XS?

Some "easy" idioms and rules-of-thumb to keep XS from becoming overwhelming

- ignores typemaps
- uses Perl datatype-specific functions from `perldoc perlguts`
- ignores most of the XSpp commands
- uses a Perl-level function wrapper to munge input/output if needed
  - ignores stack macros
  - avoids (most) issues of "mortalization"
- expands to "real" XS

Baby XS

What is Baby XS?

Some "easy" idioms and rules-of-thumb to keep XS from becoming overwhelming

- ignores typemaps
- uses Perl datatype-specific functions from `perldoc perlguts`
- ignores most of the XSpp commands
- uses a Perl-level function wrapper to munge input/output if needed
  - ignores stack macros
  - avoids (most) issues of "mortalization"
- expands to "real" XS

Baby XS

What is Baby XS?

Some "easy" idioms and rules-of-thumb to keep XS from becoming overwhelming

- ignores typemaps
- uses Perl datatype-specific functions from `perldoc perlguts`
- ignores most of the XSpp commands
- uses a Perl-level function wrapper to munge input/output if needed
  - ignores stack macros
  - avoids (most) issues of "mortalization"
- expands to "real" XS

Baby XS

What is Baby XS?

Some "easy" idioms and rules-of-thumb to keep XS from becoming overwhelming

- ignores typemaps
- uses Perl datatype-specific functions from `perldoc perlguts`
- ignores most of the XSpp commands
- uses a Perl-level function wrapper to munge input/output if needed
  - ignores stack macros
  - avoids (most) issues of "mortalization"
- expands to "real" XS

Baby XS

What is Baby XS?

Some "easy" idioms and rules-of-thumb to keep XS from becoming overwhelming

- ignores typemaps
- uses Perl datatype-specific functions from `perldoc perlguts`
- ignores most of the XSpp commands
- uses a Perl-level function wrapper to munge input/output if needed
  - ignores stack macros
  - avoids (most) issues of "mortalization"
- expands to "real" XS

Baby XS

What is Baby XS?

Some "easy" idioms and rules-of-thumb to keep XS from becoming overwhelming

- ignores typemaps
- uses Perl datatype-specific functions from `perldoc perlguts`
- ignores most of the XSpp commands
- uses a Perl-level function wrapper to munge input/output if needed
  - ignores stack macros
  - avoids (most) issues of "mortalization"
- expands to "real" XS

Types

XS has types that are like their Perl counterparts

- Scalar $\Leftrightarrow$ `SV*`
- Array $\Leftrightarrow$ `AV*`
- Hash $\Leftrightarrow$ `HV*`

Of course XS is really C so it also has types like

- `int`
- `double`
- `char*`

. . . which Perl converts to/from `SV*` when used as arguments or return value

For Future Reference

You can add you own automatic conversions via a `Typemap` file

Types

XS has types that are like their Perl counterparts

- Scalar $\Leftrightarrow$ `SV*`
- Array $\Leftrightarrow$ `AV*`
- Hash $\Leftrightarrow$ `HV*`

Of course XS is really C so it also has types like

- `int`
- `double`
- `char*`

. . . which Perl converts to/from `SV*` when used as arguments or return value

For Future Reference

You can add you own automatic conversions via a `Typemap` file

Types

XS has types that are like their Perl counterparts

- Scalar $\Leftrightarrow$ `SV*`
- Array $\Leftrightarrow$ `AV*`
- Hash $\Leftrightarrow$ `HV*`

Of course XS is really C so it also has types like

- `int`
- `double`
- `char*`

. . . which Perl converts to/from `SV*` when used as arguments or return value

For Future Reference

You can add you own automatic conversions via a `Typemap` file

Sample XS File

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

int meaning () { return 42 }
void greet (char* name)
{
  printf( "Hello %s\n", name )
}

MODULE = My::Module    PACKAGE = My::Module

int
meaning ()

void
greet (name)
    char* name
```

Sample XS File

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

int meaning () { return 42 }
void greet (char* name)
{
  printf( "Hello %s\n", name )
}

MODULE = My::Module     PACKAGE = My::Module

int
meaning ()

void
greet (name)
    char* name
```

Begin XS section

Sample XS File

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

int meaning () { return 42 }
void greet (char* name)
{
  printf( "Hello %s\n", name )
}

MODULE = My::Module     PACKAGE = My::Module

int
meaning ()

void
greet (name)
    char* name
```

XS standard inclusions
(loads standard C headers)

Begin XS section

Sample XS File

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

int meaning () { return 42 }
void greet (char* name)
{
  printf( "Hello %s\n", name )
}

MODULE = My::Module     PACKAGE = My::Module

int
meaning ()

void
greet (name)
    char* name
```

XS standard inclusions
(loads standard C headers)

C Code

Begin XS section

Sample XS File

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

int meaning () { return 42 }
void greet (char* name)
{
  printf( "Hello %s\n", name )
}

MODULE = My::Module     PACKAGE = My::Module

int
meaning ()

void
greet (name)
    char* name
```

XS standard inclusions
(loads standard C headers)

C Code

Begin XS section

XS Code
(Shown: Simple Declarations)

Using Scalars (SV*s)

SV* behave like scalars in Perl, but have different actions based on type

Creating

- `SV* newSViv(IV)`
- `SV* newSVnv(double)`
- `SV* newSVpvf(const char*, ...)`
- `SV* newSVsv(SV*)`

Accessing

- `int SvIV(SV*)`
- `double SvNV(SV*)`
- `char* SvPV(SV*, STRLEN len)`
- `char* SvPV_nolen(SV*)`

Other Actions

Plenty of other Perl-like actions, see `perldoc perlguts` for more.

- `SvTRUE(SV*)` — test for "truth"
- `sv_catsv(SV*, SV*)` — join strings

Using Scalars (SV*s)

SV* behave like scalars in Perl, but have different actions based on type

Creating

- `SV* newSViv(IV)`
- `SV* newSVnv(double)`
- `SV* newSVpvf(const char*, ...)`
- `SV* newSVsv(SV*)`

Accessing

- `int SvIV(SV*)`
- `double SvNV(SV*)`
- `char* SvPV(SV*, STRLEN len)`
- `char* SvPV_nolen(SV*)`

Other Actions

Plenty of other Perl-like actions, see `perldoc perlguts` for more.

- `SvTRUE(SV*)` — test for "truth"
- `sv_catsv(SV*, SV*)` — join strings

Using Scalars (SV*s)

SV* behave like scalars in Perl, but have different actions based on type

Creating

- `SV* newSViv(IV)`
- `SV* newSVnv(double)`
- `SV* newSVpvf(const char*, ...)`
- `SV* newSVsv(SV*)`

Accessing

- `int SvIV(SV*)`
- `double SvNV(SV*)`
- `char* SvPV(SV*, STRLEN len)`
- `char* SvPV_nolen(SV*)`

Other Actions

Plenty of other Perl-like actions, see `perldoc perlguts` for more.

- `SvTRUE(SV*)` — test for "truth"
- `sv_catsv(SV*, SV*)` — join strings

Using Arrays (AV*s) and Multiple Returns

- Perl-like functions, e.g. `av_push`
- filled with `sv*` objects
- used as argument or return value, Perl uses references
- "mortalization" problem for returns
- recommended Baby XS way to return multiple values:

```
AV* foo () {
  AV* ret = newAV();
  /* fix mortalization */
  sv_2mortal((SV*)ret);

  av_push(ret, newSViv(1));
  av_push(ret, newSVpvf("%s", "bar"));

  /* return [ 1, "bar" ] */
  return ret;
}
```

Using Arrays (AV*s) and Multiple Returns

- Perl-like functions, e.g. `av_push`
- filled with `SV*` objects
- used as argument or return value, Perl uses references
- "mortalization" problem for returns
- recommended Baby XS way to return multiple values:

```
AV* foo () {
  AV* ret = newAV();
  /* fix mortalization */
  sv_2mortal((SV*)ret);

  av_push(ret, newSViv(1));
  av_push(ret, newSVpvf("%s", "bar"));

  /* return [ 1, "bar" ] */
  return ret;
}
```

Using Arrays (AV*s) and Multiple Returns

- Perl-like functions, e.g. av_push
- filled with SV* objects
- used as argument or return value, Perl uses references
- "mortalization" problem for returns
- recommended Baby XS way to return multiple values:

```
AV* foo () {
  AV* ret = newAV();
  /* fix mortalization */
  sv_2mortal((SV*)ret);

  av_push(ret, newSViv(1));
  av_push(ret, newSVpvf("%s", "bar"));

  /* return [ 1, "bar" ] */
  return ret;
}
```

Using Arrays (AV*s) and Multiple Returns

- Perl-like functions, e.g. `av_push`
- filled with `SV*` objects
- used as argument or return value, Perl uses references
- "mortalization" problem for returns
- recommended Baby XS way to return multiple values:

```
AV* foo () {
  AV* ret = newAV();
  /* fix mortalization */
  sv_2mortal((SV*)ret);

  av_push(ret, newSViv(1));
  av_push(ret, newSVpvf("%s", "bar"));

  /* return [ 1, "bar" ] */
  return ret;
}
```

Using Arrays (AV*s) and Multiple Returns

- Perl-like functions, e.g. av_push
- filled with SV* objects
- used as argument or return value, Perl uses references
- "mortalization" problem for returns
- recommended Baby XS way to return multiple values:

```
AV* foo () {
  AV* ret = newAV();
  /* fix mortalization */
  sv_2mortal((SV*)ret);

  av_push(ret, newSViv(1));
  av_push(ret, newSVpvf("%s", "bar"));

  /* return [ 1, "bar" ] */
  return ret;
}
```

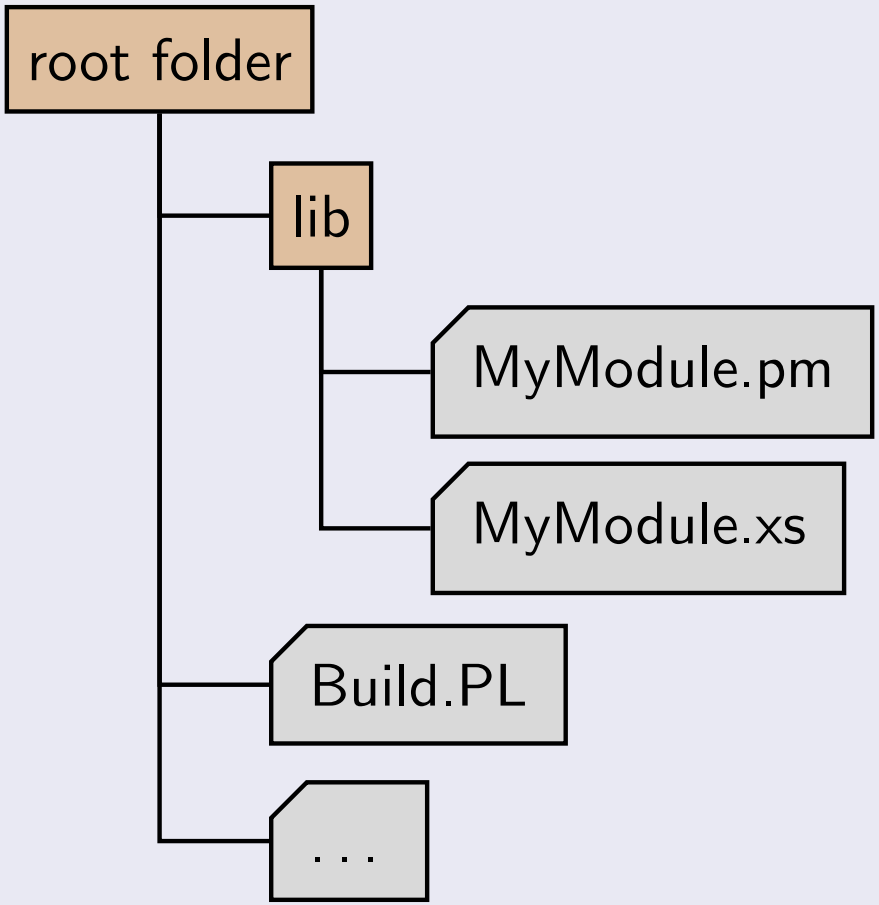
Sample Perl Module

```
package MyModule;

use strict; use warnings;
our $VERSION = '0.01';

require XSLoader;
XSLoader::load();

sub myfunc {
  my @args = @_;
  my $ret = c_func(@args);
  return wantarray ? @$ret : $ret->[0];
}
```

- Wrap C function calls in Perl subs
  - munge inputs / outputs easier
  - abstraction if C function changes
- Export the Perl function (if desired)

Building/Packaging (Module::Build)

Structure



Build.PL

```
use strict;
use warnings;

use Module::Build;

my $builder = Module::Build->new(
    module_name         => 'MyModule',
    dist_author         => 'Joel Berger',
    license             => 'perl',
    configure_requires => {
      'Module::Build' => 0.38,
    },
    build_requires     => {
      'ExtUtils::CBuilder' => 0,
    },
    extra_linker_flags => '-lsomelib',
);

$builder->create_build_script;
```

Other C Connections

There are other mechanisms for hooking into C

- `Inline::C`
  - write C in your Perl script
  - builds/loads the XS for you!
  - great for quick checks
- `PDL::PP`
  - part of `PDL`
  - special C interaction for fast numerical processing
  - has its own syntax

Acknowledgements

UIC UNIVERSITY OF ILLINOIS AT CHICAGO

- Graduate College Dean's Fellowship (Major Funding)
- LAS Ph.D. Travel Award (Conference Funding)

`https://github.com/jberger/YAPCNA2012`