Inline

or

Pathologically Polluting Perl

Andy Adler

There are some stunningly novel ideas in Perl. Many are stunningly bad, but that's always true of ambitious efforts.

Outline

- Ways to link Perl to Other Stuff
- Using Inline::C
- Writing your own Inline::
- Experiences with Inline in a large project

Why Inline?

- Isn’t Perl Perfect?
- No. “Perl” != “Perfect”
- However, “Perl” =~ /Per([fect]*)/ which is more than we can say for C, Java, Python, etc..

Linking Perl to Stuff: XS

Good:

- Powerful

Bad:

- Requires learning a new language
- Requires knowing about Perlguts, even for simple stuff
- need to create many accessory files

Simple Example

- We need to link to library add

```
$ cat add.c
#include <add.h>
int add(int a,int b) {
    return a+b;
}
$ cat add.h
int add(int a,int b);
$ gcc -shared -I. add.c -o libadd.dll
```

Running on cygwin
Need to use windows extension (or be Trickier)

Using libadd

- Using libadd from C

```
$ cat testadd.c
   #include <add.h>
   #include <stdio.h>
   int main() {
      printf("Hello World, 2+2=%d\n",
                add(2,2) );
      return 0;
   }
$ gcc -I. testadd.c add.dll -o testadd
```

- Now, run it

```
$ ./testadd.exe
   Hello World, 2+2=4
```

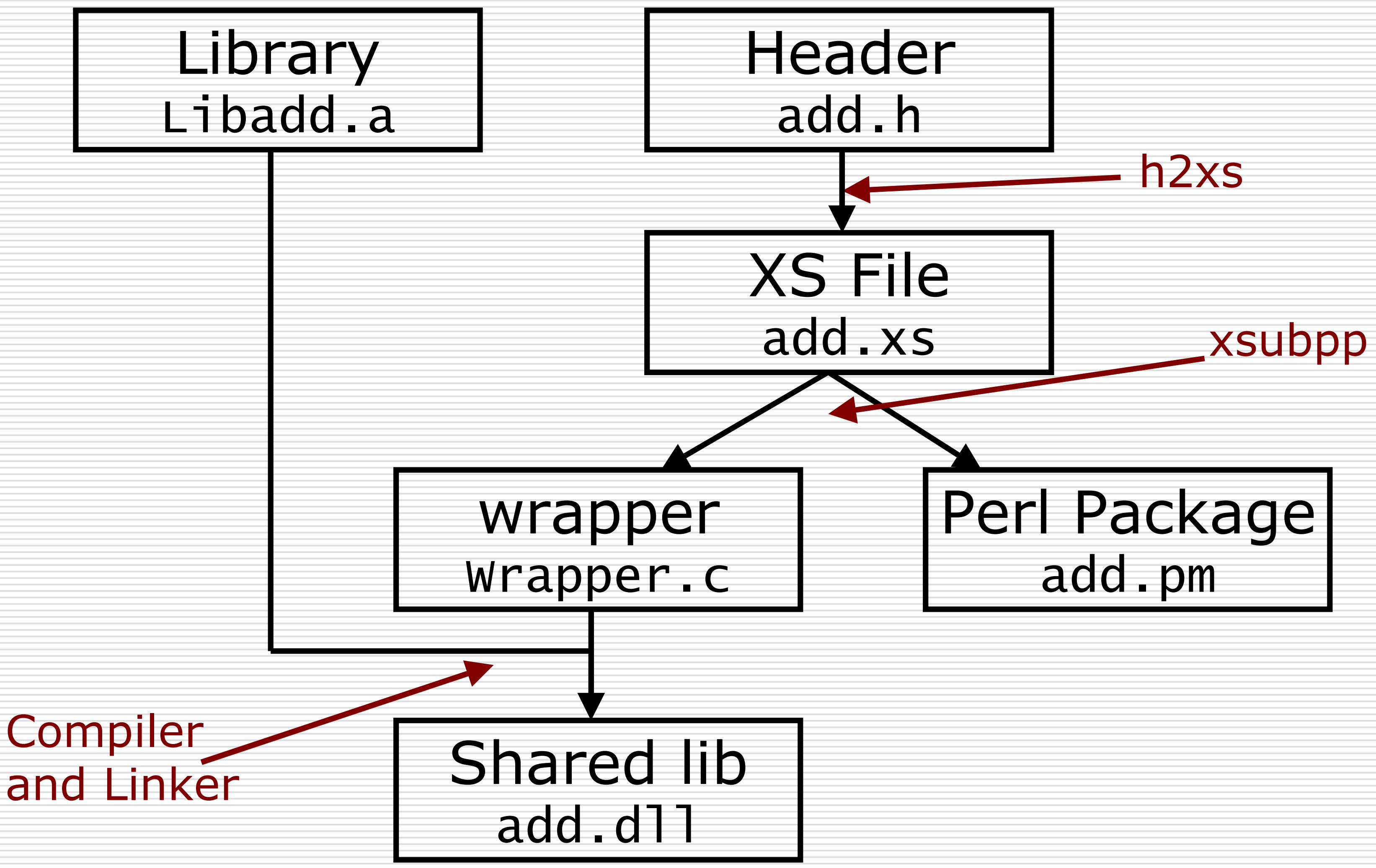
Using libadd from perl

- We want to be able to do this

```
use add.pm;
print “print 2+2”.add(2,2).”\n”;
```

- How can this be done?
  - XS
  - Swig
  - Inline

XS Approach


Library
Libadd.a
Header
add.h
h2xs
XS File
add.xs
xsubpp
wrapper
Wrapper.c
Perl Package
add.pm
Compiler
and Linker
Shared lib
add.dll

XS Example

- Use h2xs

```
$ h2xs -A -n add
   Writing add/ppport.h
   Writing add/add.pm
   Writing add/add.xs
   Writing add/Makefile.PL
   Writing add/README
   Writing add/t/1.t
   Writing add/Changes
   Writing add/MANIFEST
```

- h2xs creates the module “add”. The important file is add.xs

XS Example: add.xs

```
$ cat add/add.xs
#include "EXTERN.h"
#include "perl.h"
#include "XSUB.h"
#include "ppport.h"

MODULE = add            PACKAGE = add
int
add(a,b)
      int a
      int b
   CODE:
      RETVAL = a+b;
   OUTPUT:
      RETVAL
```

Add our own Code starting here

XS Example (cont’d)

- Build

```
$ perl Makefile.PL
$ make
mv add.dll libadd.dll.a blib/arch/auto/add/
chmod 755 blib/arch/auto/add/add.dll
Manifying blib/man3/add.3pm
```

- Test (we don’t want to make install)

```
$ perl -Iblib/arch/auto/add -Madd
   -e'print “2+2=“.add::add(2,2)'
2+2=4
```

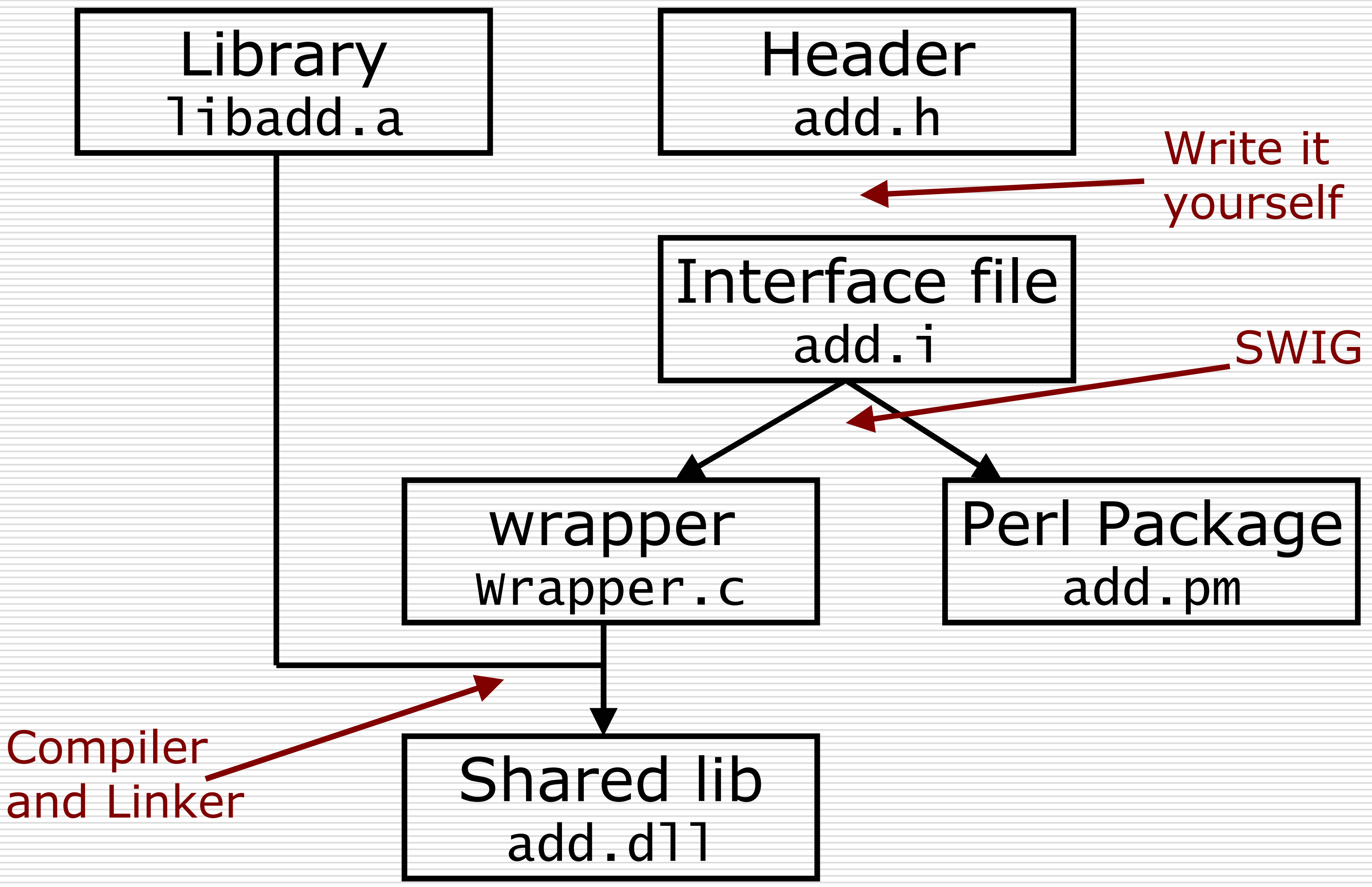
Good things about XS

- Complete access to perlguts
- Fairly straightforward for many uses
- Language is close to C

Issues with XS

- XS uses an unusual syntax
- Makefiles are autogenerated, and need to be customized
- Doing testing with the XS directory layout is awkward
- Doing anything more complicated, like linking to external libraries, requires lots more work than this example

SWIG Approach



SWIG approach

- Need to hand generate an “interface” file: add.i

```
%module add
%{
#include “add.h”
%}
#include “add.h”
```

- Then we run swig

```
swig –perl5 add.i
```

- Creates add_wrap.c
- Now we write a Makefile.PL, etc.

Pros and cons: SWIG

Good:

- Automates much of the build process

Bad:

- Requires learning a new language
- Not part of Perl distribution
- Versioning issues
  - Needed special version for windows/perl
  - Reportedly works correctly now
- Creates extra files

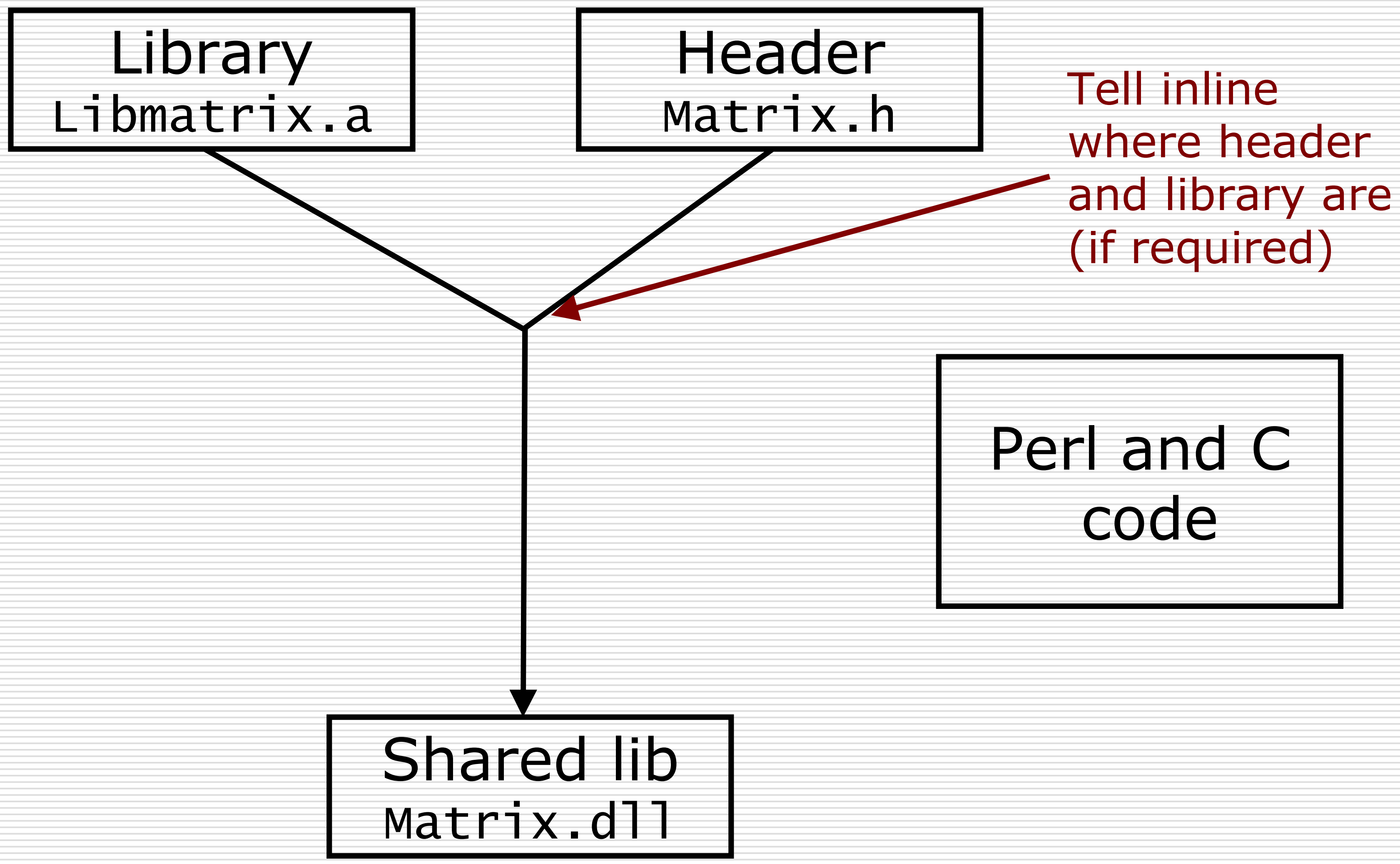
Philosophical Aside

Assertion: Creating lots of files is bad

- Many languages (notably Java) force you to create files
- However, the *raison d'être* of files is to organize information for the user. Any programming language with interferes with this is evil, evil, evil

Now, back to your regularly scheduled talk.

Inline Approach



Inline Example

Inline puts the C code right into the file

```
$ cat ex1.pl
   use Inline C;
   print "2+2=".add(2,2)."\n";
   __END__
   __C__
   int add(int a,int b) {
      return a+b;
   }
$ perl ex1.pl
   2+2=4
```

Linking Perl to Stuff: Inline

Good:

- Very DWIM (Takes care of details for you)
- Almost no unnecessary syntax
- Easy to learn
- Can write *One liners* with Inline

Bad:

- Not as powerful as XS
- Can’t distribute modules (easily) without Inline (long winded approach as of 0.40)

Installing Inline

- Requirements

  Parse::RecDescent

  (Inline 0.44 doesn't require it, but it's to build it any other way)

- Build
  - Standard way: perl Makefile.PL ; make
  - Warning: make test fails on cygwin perl – but in installs/works fine anyway

Using Inline::C

CODE:

```
use Inline C => <<'END_C';
void greet(char *greetee) {
  printf("Hello, %s\n“,greetee);
}
END_C
greet(“world”);
```

OUTPUT:

```
Hello, world
```

Inline figured out how to bind perl strings to char *

Using Inline::C

CODE:

```
use Inline C;
print JAxH('Perl‘);
__END__
__C__
SV* JAxH(char* x) {
   return newSVpvf(
    "Just Another %s Hacker\n", x);
}
```

Optionally, we could return a char * pointer to a static buffer

OUTPUT:

```
Just Another Perl Hacker
```

Inline Use: (Win2K ActivePerl)

```
$ TIMEFORMAT="Time= %R"
$ time C:/perl/bin/perl ex2.pl
Just Another Perl Hacker
Time= 6.743
$ time C:/perl/bin/perl ex2.pl
Just Another Perl Hacker
Time= 0.239
```

Inline Directories

```
drwxr-xr-x         0 Nov  4 20:42 _Inline
-rw-r--r--       135 Nov  4 20:39 ex2.pl
./_Inline:
drwxr-xr-x         0 Nov  4 20:42 build
-rw-r--r--       221 Nov  4 20:42 config
drwxr-xr-x         0 Nov  4 20:42 lib
./_Inline/build:
./_Inline/lib:
drwxr-xr-x         0 Nov  4 20:42 auto
./_Inline/lib/auto:
drwxr-xr-x         0 Nov  4 20:42 ex2_pl_1031
./_Inline/lib/auto/ex2_pl_1031:
-r--r--r--         0 Nov  4 20:42 ex2_pl_1031.bs
-r-xr-xr-x     20480 Nov  4 20:42 ex2_pl_1031.dll
-r--r--r--       832 Nov  4 20:42 ex2_pl_1031.exp
-rw-r--r--       594 Nov  4 20:42 ex2_pl_1031.inl
-r--r--r--      2234 Nov  4 20:42 ex2_pl_1031.lib
```

Inline Syntax

- Usage:

```
use Inline Language => source-code,
    config_option => value,
    config_option => value;
```

- source-code
  - String: `use Inline C => q{ . . . };`
  - Here Doc: `use Inline C => <<END_C;`
  - __END__: `use Inline C;`

Inline Syntax

Configuration Options:

- Using external libraries:

Link to libmylib.a or mylib.lib

```
use Inline C =>
   DATA =>
     LIBS => '-lmylib',
     PREFIX => 'my_';
```

Warning

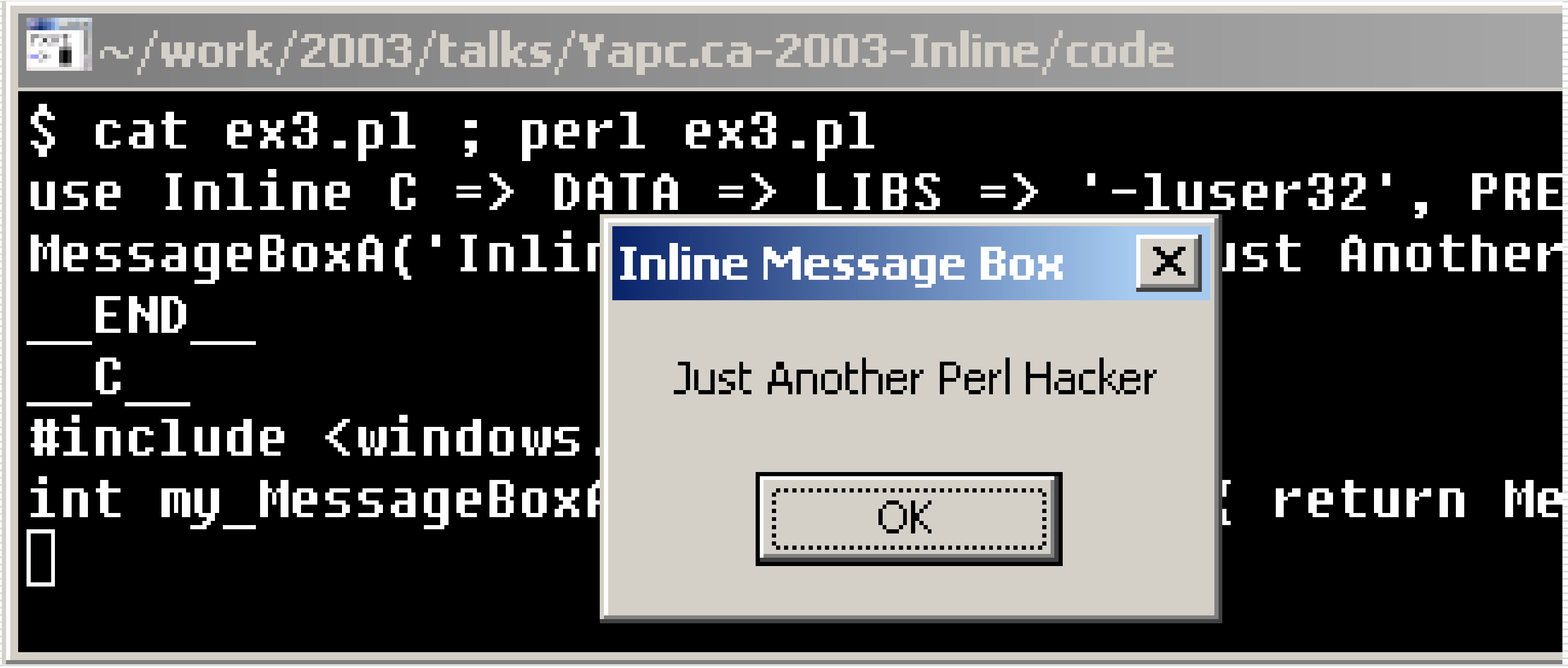
The next slides contain windows specific code.

Viewer discretion is advised

External Libraries

```
use Inline C => DATA => LIBS =>
   '-luser32', PREFIX => 'my_';
MessageBoxA('Inline Message Box',
   'Just Another Perl Hacker');
__END__
__C__
#include <windows.h>
int my_MessageBoxA(char* C, char* T){
   return MessageBoxA(0, T, C, 0); }
```

External Libraries



~/work/2003/talks/Yapc.ca-2003-Inline/code

```
$ cat ex3.pl ; perl ex3.pl
use Inline C => DATA => LIBS => '-luser32', PRE
MessageBoxA('Inlin                          ust Another
__END__
__C__
#include <windows.
int my_MessageBoxA                         { return Me
```

Just Another Perl Hacker

See Perl Run. Run Perl, Run!

Inline::CPR -> Create C interpreter

```
#!/usr/bin/cpr
int main(void) {
  printf("Hello, world\n");
}
```

See Perl Run even more

- Call perl from C code

```
#!/usr/bin/cpr
int main(void) {
   printf("I'm running under "
            "Perl version %s\n",
      CPR_eval("use Config; "
                 "$Config{version}"));
   return 0;
}
```

- Running this program prints:

```
I'm running under Perl version 5.8.0
```

CPR behind the covers

- /usr/bin/cpr wraps c code like this

```
BEGIN{ mkdir('./_cpr', 0777) unless -d
   './_cpr';}
use Inline Config => DIRECTORY => './_cpr/';
use Inline CPR;
cpr_main();
__END__
__CPR__
int main(void) {
    printf("Hello World");
    return 42;
}
```

CPR issues

- Just before the talk I was trying to get cpr working on cygwin
- However, cygwin has a bizarre notion of execute permissions
  - *.exe, *.bat etc have it, others don’t
- Fix this by doing
  - pl2bat /usr/bin/cpr.pl
  - And some minor edits to cpr.bat
- Now use #!/usr/bin/cpr.bat

Other Inline Languages

ILSM = Inline Language Support Module

Inline::CPP
Acme::Inline::PERL
Inline::Java
Inline::Guile
Inline::C
Inline::Befunge
Inline::BC
Inline::TT
Inline::WebChat
Inline::Ruby

Inline::Tcl
Inline::Python
Inline::Pdlpp
Inline::Octave
Inline::Basic
Inline::Filters
Inline::Awk
Inline::ASM
Inline::Struct

Creating an Inline Module

Techniques to link to Perl

- Compile to a dynamic library (*.so,*.dll) and link to Perl at run time (::C, ::CPP, ::Java::API)
- Open a socket connection between Perl and the other interpreter (::Python, ::Java)
- Pipe stdio,stderr between Perl and other interpreter (::Octave). (using IPC::Open3)

Inline::Python example

- $ cat ex_python.pl

```
use Inline Python;
my $language = shift;
print $language, (match($language, 'Perl')
      ? ' rules' : ' sucks'), "!\n";
__END__
__Python__
import sys
import re
def match(str, regex):
   f = re.compile(regex);
   if f.match(str): return 1
   return 0
```

- Test: ./ex_python.pl Perl
  - Perl rules,   or
  - Python sucks

How to create Inline Module

Look at Inline::PERL (or Inline-API)

Inline::PERL gives you the power of the PERL programming language from within your Perl programs. …

PERL is a programming language for writing CGI applications. It's main strength is that it doesn't have any unnecessary warnings or strictures.

Create Inline Module

- Create the following methods
  - Register
  - Build
  - Load
  - Validate
- Object variables contains all the code and administrative information

Inline::Octave

- Sample code

```
use Inline Octave ;
$c= new Inline::Octave::Matrix(
   [ [1.5,2,3],[4.5,1,-1] ]);
my $d= addone($c) x $c->transpose;
print $d->disp;
oct_plot( [0..4], [3,2,1,2,3] );
sleep(5);
__DATA__
__Octave__
function x=addone(u); x=u+1; endfunction

## Inline::Octave::oct_plot (nargout=0)  => plot
```

Custom Octave function

Don't have Access to number of Output params From perl5

Example of a “load” method

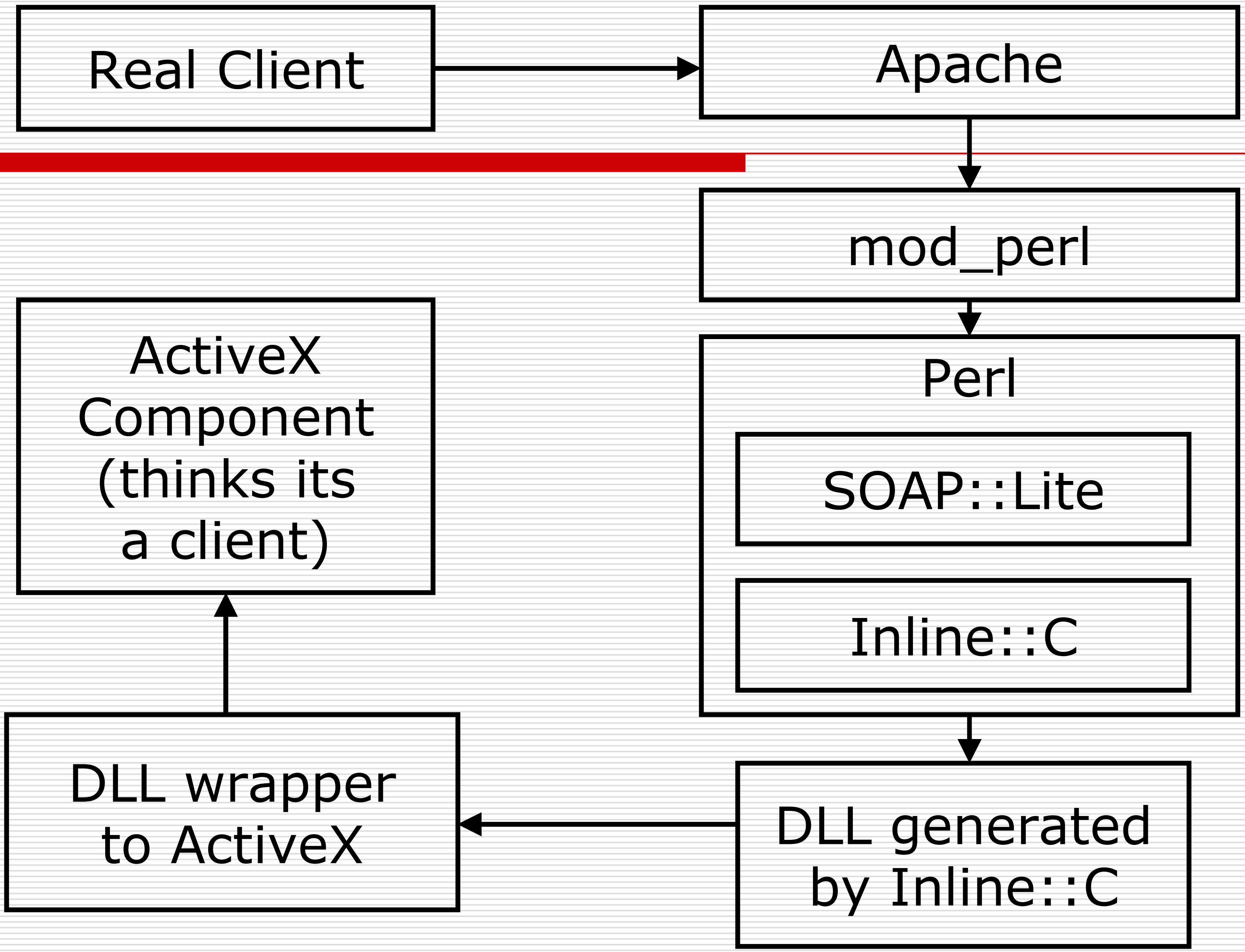
```
sub load {
    my $o = shift;
    my $obj = $o->{API}{location};
    open PERL_OBJ, "< $obj" or croak
        "Can't open $obj for output\n$!";
    my $code = join '', <PERL_OBJ>;
    close \*PERL_OBJ;
    eval
        "package $o->{API}{pkg};\n$code";
    croak "$obj:\n$@" if $@;
}
```

Current Status of Inline

- Version 0.44 was recently released
  - New, cleaner build
  - Bug fixes
  - New parser (Regexp instead of RecDescent)
- Version 0.50 promises:
  - Distribute modules without Inline
  - Cleaner features

Real world example

- We had a biometric software compiled as a client side ActiveX
- We needed it to run as a server component with a SOAP protocol
- Tasks
  - Convince the ActiveX to work like a regular DLL
  - Get the DLL to talk to SOAP::Lite running in mod_perl in Apache


Real Client
Apache
mod_perl
Perl
SOAP::Lite
Inline::C
ActiveX Component (thinks its a client)
DLL wrapper to ActiveX
DLL generated by Inline::C

Code Design

- Modules:
  - BiometricInterface.pm
  - SpecificVendor.pm
- Modifications to Apache Config
  - set perl @INC
  - Can’t use directory with spaces in path – Inline can’t handle this
  - DLLs were put in C:/Apache directory, so that perl would pick them up

SOAP::Lite in mod_perl

- Entry point *.cgi

```
use BiometricInterface;
use SpecificVendor;
SOAP::Transport::HTTP::CGI  ->
   dispatch_to(‘SpecificVendor')
      -> handle;
```

SpecificVendor Module

```
use Inline C => Config => MYEXTLIB =>
   "C:/Apache/SpecificVendor.lib";
use Inline C => <<END_OF_C_CODE;
#define DLLEXPORT __declspec(dllimport) __cdecl
DLLEXPORT int InitEngine() ;
DLLEXPORT int CreateTemplate( char * infile,
                              char * outfile );
int init_engine() {
   return InitEngine();
}

int create_template( char * infile,
                     char * outfile ) {
   return CreateTemplate( infile, outfile );
}
```

Comments

- Inline (and SOAP::Lite) allowed the proof of concept to done very quickly (1 month)
- Eventually it was replaced with a JSP + JAVA native interface application
- Inline is somewhat crude when emdedded. Its hard to distribute
  - All paths are absolute
  - Inline module must be included
  - If file timestamps change, system looks for a compiler to recompile code
  - End up writing lots of thin wrappers

Summary

- Ways to link Perl to Other Stuff
  - XS
  - SWIG
  - Inline
- Using Inline::C
- Writing your own Inline::
- Experiences with Inline in a large project

Comments?

References:

- Pathologically Polluting Perl,
  Brian Ingerson, Feb. 06, 2001, www.perl.com
- Inline-FAQ, www.cpan.org

Inline with Perl. YAPC::CA 2003. Andy Adler

Perl's Inline module is a wonderful idea: write stuff in Perl that is is best done in Perl, and write stuff in other languages as required. The Inline module will then fit the bits together, converting between native types in each language as required. Of course, being Perl, there are many ways to do it. The official way to mix Perl and C is XS, and several other solutions (such as SWIG) exist as well. Inline has several key advantages:

- Simplicity: it tries vary hard to "DWIM" (do what I mean).
- Simplicity: no need to create Makefiles
- Simplicity: all languages use the same Inline syntax and
- Obfuscability: you can create One liner JAPH's with it.
- We'll look at using Inline for C code using (surprise) Inline::C. We'll play with interpreted C code with Inline::CPR. And we'll look at (some aspects of) writing an Inline::* module.