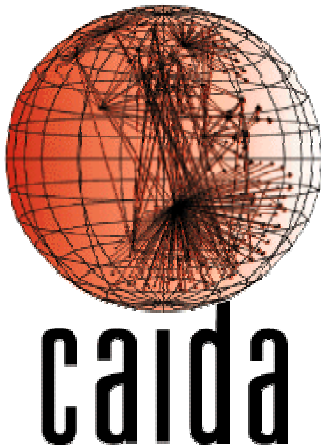


Embedding C/C++ in Perl

David Moore



CAIDA
University of California, San Diego

Perl Mongers – November 12, 2007



Embedding C/C++ in Perl

- About: embedding C/C++ in Perl
 - Use existing libraries
 - Speed up portions of perl code
 - Provide both Perl and C/C++ interfaces for your libraries
- Not about: embedding Perl interpreter into other applications

Approaches

- XS
 - traditional perl approach
 - swig & Inline::* use it under the covers
- SWIG – Simplified Wrapper and Interface Generator
 - <http://www.swig.org/>
- Inline::C/C++
 - CPAN (note that <http://inline.perl.org/> is DEAD)
 - <http://www.mail-archive.com/inline@perl.org/>
- All support unix and windows.

Approaches

- SWIG

- C/C++ → perl, ruby, python, tcl, guile, R, java (JNI), ...
- requires **ONLY** header files
- necessary files can be pre-generated and shipped with package, so users don't need swig installed
- not designed for dynamic C/C++ uses

- Inline::*

- perl ← C, C++, Java, ...
- requires header files **AND** source code
- users need Inline::* installed, but building process may be more transparent to them
- dynamically make C/C++ code inside perl and use it

SWIG

- Latest versions:
 - use 1.3.X tree
 - 1.1p5 installed many places, but quite old in functionality
- Mailing list and community is quite active
- Heavily documented
- Knows STL types, can handle templates
- Doesn't handle nested classes
- Real parser
- Fixes can live in dev tree a long time before main release
- Naming and design decisions can change in the dev tree

Inline::C and Inline::C++ (CPP)

- Latest versions on CPAN:
 - Inline::C 0.44 (November 2002)
 - Inline::CPP 0.25 (June 2001)
- Mailing list does have some activity (mostly about Java)
- Some bugs/omissions in Inline::C++ caused me serious problems and some I had to patch source
 - Optional/default arguments are given off by one position value. e.g., `void foo(int x, int y=1);` perl: `foo(10);` → `foo(10,10);`
 - Parser did not accept 'virtual' in front of constructor or destructor, which is required for proper handling of certain derived classes.
- Regexp based parser, confused by certain constructs
 - `#ifdef`, `#if 0`, etc...
 - no templates, no functions taking void arguments, no ...

Example Code

```
/* File : example.c */
#include <time.h>

double My_variable = 3.0;

int my_mod(int x, int y) {
    return (x%y);
}

char *get_time() {
    time_t ltime;
    time(&ltime);
    return ctime(&ltime);
}
```

Example in Inline::C

```
#!/bin/perl

use Inline `C`;

print my_mod(107,10), "\n";

__END__
__C__
/* Include the code right inside your .pl or .pm! */
#include <time.h>

double My_variable = 3.0; /* Not accessible from perl */

int my_mod(int x, int y) {
    return (x%y);
}

char *get_time() {
    time_t ltime;
    time(&ltime);
    return ctime(&ltime);
}
```

Example in SWIG

```
/* example.i (could be easily based on example.h) */
%module example
%{
    /* Put header files here or function declarations
    like below */
extern double My_variable;
extern int fact(int n);
extern int my_mod(int x, int y);
extern char *get_time();
%}

extern double My_variable;
extern int fact(int n);
extern int my_mod(int x, int y);
extern char *get_time();
```

Example in SWIG

- `swig -perl5 example.i`
 - generates `example_wrap.c` and `example.pm`
- `gcc -c example.c example_wrap.c \`
`perl -MExtUtils::Embed -e ccopts \`
 - builds `example.o` and `example_wrap.o`
- `ld -G example.o example_wrap.o -o example.so`
 - generate shared library
- use in perl:

```
use example;
print $example::My_variable, "\n";
print example::my_mod(107,10), "\n";
```

Overall

- Ok, you can't go too wrong with SWIG, however here are some situations where you might prefer Inline::C/C++:
 - already existing perl code that you want to replace with optimized versions
 - have no desire to use the code separately from your module
 - less mental/organizational startup time and “quicker” edit-test cycle