TITLE

Pathologically Polluting Perl with C, Python and Other Rubbish using Inline.pm

---

Introduction

No programming language is Perfect. Perl comes very close. **P**! **e**! **r**! *l*? :-(Not quite ‘‘Perfect’’. Sometimes it just makes sense to use another language for part of your work. You might have a stable, pre-existing code base to take advantage of. Perhaps maximum performance is the issue. Maybe you just ‘‘know how to do it’’ that way. Or very likely, it’s a project requirement forced upon you by management. Whatever the reason, wouldn’t it be great to use Perl most of the time, but be able to invoke something else when you had to?

`Inline.pm` is a new module that glues other programming languages to Perl. It allows you to write C, C++, and Python code directly inside your Perl scripts and modules. This is conceptually similar to the way you can write inline assembly language in C programs. Thus the name: `Inline.pm`.

The basic philosophy behind Inline is this: ‘‘make it as easy as possible to use Perl with other programming languages, while ensuring that the user’s experience retains the DWIMity of Perl’’. (1) To accomplish this, Inline must do away with nuisances such as interface definition languages (2), makefiles, build directories and compiling. You simply write your code and run it. Just like Perl.

Inline will silently take care of all the messy implementation details and ‘‘do the right thing’’. It analyzes your code, compiles it if necessary, creates the correct Perl bindings, loads everything up, and runs the whole schmear. The net effect of this is you can now write functions, subroutines, classes, and methods in another language and call them as if they were Perl.

Inline in Action - Simple examples in C

Inline addresses an old problem in a completely revolutionary way. Just describing Inline doesn’t really do it justice. It should be *seen* to be fully appreciated. Here are a few examples to give you a feel for the module.

Hello, world

It seems that the first thing any programmer wants to do when he learns a new programming technique is to use it to greet the Earth. In keeping with that tradition, here is the ‘‘Hello, world’’ program using Inline.

```
    use Inline C => <<’END_C’;
    void greet() {
        printf("Hello, world\n");
    }
    END_C

    greet;
```

Simply run this script from the command line and it will print (you guessed it):

```
    Hello, world
```

In this example, `Inline.pm` is instantiated with the name of a programming language, ‘‘C’’, and a string containing a piece of that language’s source code. The source code string is represented using the here-document quoting style, for clarity. This C code defines a function called `greet()` which gets bound to the Perl subroutine `&main::greet`. Therefore, when we call the `greet()` subroutine, the program prints our message on the screen.

You may be wondering why there are no `#include` statements for things like `stdio.h`? That’s because Inline::C automatically appends the following lines to the top of your code:

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

These header files include all of the standard system header files, so you almost never need to use `#include` unless you are dealing with a non-standard library. This is in keeping with Inline’s philosophy of making easy things easy. (I borrowed that one :-)

Just Another ____ Hacker

The next logical question is, ‘‘How do I pass data back and forth between Perl and C?’’ In this example we’ll pass a string to a C function and have it pass back a brand new Perl scalar.

```
    use Inline C;
    print JAxH(’Perl’);


    __END__
    __C__
    SV* JAxH(char* x) {
        return newSVpvf("Just Another %s Hacker\n", x);
    }
```

When you run this program, it prints:

```
    Just Another Perl Hacker
```

You’ve probably noticed that this example is coded differently then the last one. The `use Inline` statement specifies the language being used, but not the source code. This is an indicator for Inline to look for the source at the end of the program. (More about this later.)

The concept being demonstrated is that we can pass Perl data in and out of a C function. Using the default Perl type conversions, Inline can easily convert all of the basic Perl data types to C and vice-versa.

This example uses a couple of the more advanced concepts of Inlining. Its return value is of the type `SV*` (or Scalar Value). The Scalar Value is the most common Perl internal type. Also, the Perl internal function `newSVpfv()` is called to create a new Scalar Value from a string, using the familiar `sprintf()` syntax. These topics will be addressed again later. (See It Takes All Types and Simple Perl Internals below.)

Do The Math

The previous examples only had one C function each. Let's look at a program that has several functions and see how they interact.

```
    print "9 + 5 = ", add(9, 5), "\n";
    print "SQRT(9^2 + 5^2) = ", pyth(9, 5), "\n";
    print "9 * 5 = ", mult(9, 5), "\n";


    use Inline C => <<'END_C';
    int add(int x, int y) {
        return x + y;
    }
    static int mult(int x, int y) {
        return x * y;
    }
    double pyth(int x, int y) {
        return sqrt(add(mult(x, x), mult(y, y)));
    }
    END_C
```

This produces:

```
    9 + 5 = 14
    SQRT(9^2 + 5^2) = 10.295630140987
    Can't locate auto/main/mult.al in @INC ...
```

This is definitely the hard way to do simple math, but it reveals some important concepts about Inline C. The first is that Inline functions like `add()` can be called either from Perl or C. Secondly, standard C functions like `sqrt()` can be called without any fuss, because they are already included in Perl. Finally, if you declare a function to be `static`, it won't be visible from Perl. Thus the error message.

What about XS and SWIG?

Let's detour momentarily to ponder ‘‘Why Inline?’’

There are already two major facilities for extending Perl with C. They are XS and SWIG. Both are similar in their capabilities, at least as far as Perl is concerned. And both of them are quite difficult to learn compared to Inline. Since SWIG isn't used in practice to nearly the degree that XS is, I'll only address XS.

There is a big fat learning curve involved with setting up and using the XS environment. You need to get quite intimate with the following docs:

```
 * perlxs
 * perlxstut
 * perlapi
 * perlguts
 * perlcall
 * perlmod
 * h2xs
 * xsubpp
 * ExtUtils::MakeMaker
```

With Inline you can be up and running in minutes. There is a C Cookbook with lots of short but complete programs that you can extend to your real-life problems. No need to learn about the complicated build process going on in the background. You don't even need to compile the code yourself. Perl programmers cannot be bothered with silly things like compiling. ‘‘Tweak, Run, Tweak, Run’’ is our way of life. Inline takes care of every last detail except writing the C code.

Another advantage of Inline is that you can use it directly in a script. As we'll soon see, you can even use it in a Perl one-liner. With XS and SWIG, you always set up an entirely separate module. Even if you only have one or two functions. Inline makes easy things easy, and hard things possible. Just like Perl.

One of Inline's major goals is to provide a reasonable replacement for XS. This should soon be the case for all but the most esoteric cases.

Finally, Inline supports several programming languages (not just C and C++). As of this writing, Inline has support for C, C++, Python, and CPR. There are plans to add many more.

One-Liners

Perl is famous for its one-liners. A Perl one-liner is generally a (reasonably) short piece of Perl code that can be typed at the command line, and can successfully accomplish a task that would take much longer in another language. It is one of the popular techniques that Perl hackers use to flex their programming muscles. Here are a couple of the more famous ones:

```
     perl -le 'while(($_||=1)++){print if(1x$_)!~/^(11+)\1+$/}'
```

Prints all of the prime numbers. (3)

```
     perl -e 'require DynaLoader;DynaLoader::dl_install_xsub("main::hangme",unpack("
```

Will use the ‘‘F00F Pentium bug’’ to crash your computer. (4)

There is a special flavor of Perl one-liners that often show up as the signatures of `comp.lang.perl.*` usenet postings. This brand of one-liner is known as a JAPH. The goal of a JAPH is to print ‘‘Just Another Perl Hacker\n’’, using some obscure, insidious, morally impure programming technique. Case in point (3):

```
    perl -e ’BEGIN{my$x="Knuth heals rare project\n";$^H{integer}=sub{my$y=shift;$_=
```

So the only question that remains is this: ‘‘Is Inline powerful enough to produce a JAPH one-liner that is also bonifide C extension?’’ Of course it is! (Why else would I be going to all this trouble?) Here you go:

```
    perl -e ’use Inline C=>q{void J(){printf("Just Another Perl Hacker\n");}};J’
```

Try doing that with XS. We can even write the more complex Inline `JAxH()` discussed earlier as a one-liner:

```
    perl -le ’use Inline C=>q{SV*JAxH(char*x){return newSVpvf("Just Another %s Hacke
```

If you participate on the Inline mailing list (inline@perl.org) you’ll find that this JAPH is indeed my personal email signature. I thought this was pretty cool until Bernhard Muenzer posted this gem to `comp.lang.perl.modules`:

```
    #!/usr/bin/perl -- -* Nie wieder Nachtschicht! *- -- lrep\nib\rsu\!#
    use Inline C=>’void C(){int m,u,e=0;float l,_,I;for(;1840-e;putchar((++e>907
     &&942>e?61-m:u)["\n)moc.isc@rezneumb(rezneuM drahnreB"]))for(u=_=l=0;79-(m
      =e%80)&&I*l+_*_<6&&26-++u;_=2*l*_+e/80*.09-1,l=I)I=l*l-_*_-2+m/27.;}’;&C
```

Supported Platforms for C

Inline C works on all of the Perl platforms that I have tested it with so far. This includes all common Unixes and recent versions of Microsoft Windows. The only catch is that you must have the same compiler and `make` utility that was used to build your `perl` binary.

Unix Support

Inline has been successfully used on Linux, Solaris, AIX, HPUX, and all the recent BSDs (Free, Open, and Net).

Options for Windows Users

There are two common ways to use Inline on MS Windows. The first one is with ActiveState’s ActivePerl for MSWin32 (5). In order to use Inline in that environment, you’ll need a copy of MS Visual C++ 6.0. This comes with the `cl.exe` compiler and the `nmake` make utility. Actually these are the only parts you need. The visual components aren’t necessary for Inline.

The other alternative is to use the Cygwin utilities. (6) This is an actual Unix porting layer for Windows.

It includes all of the most common Unix utilities, such as `bash`, `less`, `make`, `gcc` and of course `perl`.

Unsupported Options

If you are stuck on a planet where you just can't get access to the right compiler, you can at least *try* to use an alternate one. The Inline C configuration allows you to specify your own overrides for the compiler, linker, and make utility, with the `CC`, `LD` and `MAKE` options. Using these, you may be able to get some alternate configurations to work, but please don't complain (to me at least :-) if it doesn't.

The Inline Syntax

Inline is a little bit different than most of the Perl modules that you are used to. It doesn't import any functions into your namespace and it doesn't have any object oriented methods. Its entire interface is specified through `'use Inline ...'` commands.

This section will explain all of the different ways to `use Inline`.

use Inline ...

You may have several `'use Inline ...'` statements in a single source file. Any configuration properties specified in one statement will be propagated to subsequent statements, if it makes sense.

Here is a list of the most common usages. Each example is followed by a short explanation.

```
    use Inline;
```

Does nothing except load the Inline module.

```
    use Inline C => "source code";
```

Compiles, binds and loads a module written in C. In this example the source code is passed in as a string.

```
    use Inline C => "source code",
               LIBS => '-lfoo',
               PREFIX => 'foo_';
```

Same as before. Configuration options are specified as (key => value) pairs.

```
    use Inline C => DATA => config-pair-list;
```

Automatically search for the C source code in Perl's special `DATA` filehandle. You can optionally specify configuration options.

```
    use Inline C;
```

This is shorthand for `'use Inline C => DATA;'` with no configuration arguments.

```
    use Inline C => [ list of source code lines ];
```

Alternatively you can pass in the source code as an anonymous array of lines.

```
    use Inline C => '/path/source.c';
```

You can also put the source code in a separate file and specify the file's name.

```
    use Inline Config =>
               DIRECTORY => '/mypath/.Inline';
```

This does not compile any code. It just sets general configuration options that will apply to subsequent `use Inline` commands.

```
    use Inline C => Config =>
               LIBS => '-lfoo';
```

If you want to specify config options for a specific language without compiling any code, use this syntax form.

```
    use Inline with => Event;
```

You can tell Inline to get hints from other Perl modules like `Event.pm`.

```
    use Inline qw(FORCE NoClean info)
```

You can specify a list of shortcut options to apply. See section on ‘‘Shortcuts’’ below.

The DATA section

Inline has a special feature designed to make your code easier to read and maintain. Normally you pass code to Inline as a string. But you can also tell Inline to search the contents of the `DATA` filehandle for source code. Consider this snippet:

```
    use Inline Lisp => 'DATA';
    # Perl stuff ...
    __END__
    __Lisp__
    ; Lisp stuff
```

The `__END__` statement in Perl marks the end of the actual Perl code. (`__DATA__` does the same thing for modules.) Everything after that marker is made into a pseudo-file that you can read using the `<DATA>` filehandle.

Inline takes advantage of this feature. By using the word 'DATA' for your source code string, you are telling Inline to read the `DATA` filehandle. Inline will scan through the text, looking for its own special marker, `__Foo__`, where ‘‘Foo’’ is the programming language you are trying to compile.

The observant hacker will see a slight problem here. Perl handles `use` statements at compile time, but the `DATA` filehandle is not available until the entire program has been compiled. How could it be? <rhetorical> When you use the `DATA` syntax, Inline queues up the request for delayed processing.

Luckily Perl has another handy construct, the `INIT` block. This is very similar to a `BEGIN` or `END` block. Perl invokes an `INIT` block in that magical moment between compilation and run time. The *Inline* `INIT` block then compiles any objects that were queued.

Objects? Yes, you can put more than one source file in the DATA area. The only rule to remember is that ‘‘order counts’’. You must have the `use` calls and the code blocks in the same order. Like this:

```
    use Inline C;
    use Inline 'C++';
    use Inline Python';
    # Perl treasures ...
    __END__
    __C__
    /* C garbage */
    __C++__
    // C++ trash
    __Python__
    """ Python pollution """
```

As was stated above:

```
    use Inline Foo;
```

is simply shorthand for:

```
    use Inline Foo => 'DATA';
```

This is fine for scripts, but module authors often reserve the `DATA` section for POD and AutoLoader subroutines. Don’t worry. Be happy. Inline will only process the DATA between its `__Foo__` type markers and POD commands like `=pod`. That way you can mix all three together.

There is one more advantage to using the `DATA` form of Inline as opposed to the string form. You don’t need to worry about escaping special characters. A backslash is a backslash is a backslash.

The `bind()` function

I lied. Inline actually does have one function (or class method). It is the `bind()` function. It allows you to compile/bind/load an extension at run time. Remember, Perl’s `use` command is a compile time directive. But imagine that you want to write a program that generates C code on the fly. (And compile and run it as well). You would not have the source code available until run time. With `bind()` you can do this:

```
    use IBM::Mainframe;
    use Inline;
    my $source_code = COBOL_generator("ebcdictate");
    Inline->bind(COBOL => $source_code);
    IBM::Mainframe::crunch(ebcdictate("Just Another Perl Hacker")
      or die "Does not compute";
```

`bind()` takes most of the same parameters that `'use Inline ...'` does, but since it’s a method call, you need to put them in parentheses.

The `init()` function

I lied again. Inline actually has **two** public functions. (Now you know why I call it ‘‘*Pathologically* Polluting Perl’’) There is also an `init()` function to handle a somewhat esoteric case.

Imagine for a moment that you wanted to write the following code:

```
eval "use Inline Java";
__END__
__Java__
/* Java junk */
```

There is a big problem here. Remember that this syntax tells Inline to look for the Java source code in the DATA area, and this actually gets done in Inline’s `INIT` block. But since we’re evaling the call, it’s already too late to run the `INIT`.

This is where `init()` comes to the rescue. It calls Inline’s `INIT` routine manually, thus compiling any Inline objects that have been queued for processing. Here is the correct code:

```
eval "use Inline Java";
Inline->init;
__END__
__Java__
/* Your Java code */
```

Configuration Options

We have already seen some examples of passing configuration information to Inline. This is accomplished via a list of ’key’ => ’value’ pairs that follow the source code parameter. Here is an example:

```
use Inline C => ’DATA’,
           DIRECTORY => ’/mypath/mydir’,
           INC => ’-I/inc/path’;
```

This tells Inline to use a specific directory to do its dirty work, and to use a non-standard path to find header files. Some config parameters can have multiple values. It that’s the case, you always list them in an anonymous array. Most parameters are also additively inherited from previous Inline calls. Another example will help make sense of that statement:

```
use Inline Config =>
           DIRECTORY => ’/mypath/mydir’;
use Inline C => Config =>
           INC => [’-I/inc/path1’, ’-I/inc/path2’];
use Inline C => ’DATA’,
           INC => ’-I/inc/path3’;
use Inline C => ’DATA’,
           DIRECTORY => ’/mypath/otherdir’,
           INC => [undef, ’-I/inc/path3’, ’-I/inc/path2’];
```

The first call sets the build/install directory for future calls to use. It is a ‘‘configuration only’’ call; no code is compiled. The second call specifies C-specific options and is also ‘‘configuration only’’. The third call compiles some C code, adding a third include path to the original two. The last call compiles some more C code. It overrides the previous DIRECTORY option. It also uses `undef` to clear the current list of include paths, then re-adds paths 3 and 2 in that order.

Most of the configuration parameters for `Inline C` are simply proxies for identical parameters in the underlying XS and MakeMaker processing. If you are already familiar with these options, that’s great. If not, no worries. You can learn them as you go. I won’t list all of them here, but suffice it to say, ‘‘Inline is very configurable’’.

Shortcuts

Say that you want to recompile your Inline C function even though its up to date. Maybe you want to peek at what Inline generated to make it all work. If your program is called `foo.pl`, then try running it like this:

```
    perl -MInline=FORCE,NOCLEAN,INFO foo.pl
```

This will *force* a recompile, leave the build area intact, and print some information telling you where to find it. `INFO`, `FORCE` and `NOCLEAN` are known as Inline **shortcuts**. They are configuration options that you can use from the command line. If you need to use them a lot you can also put them directly in your program like this:

```
    use Inline qw(FORCE NOCLEAN INFO);
```

If you ever find a bug with Inline, you can report it by saying:

```
    perl -MInline=REPORTBUG foo.pl
```

and following the instructions.

Fine Dining - A Tour of the C Cookbook

In the spirit of the O’Reilly book ‘‘Perl Cookbook’’, Inline provides a manpage called C-Cookbook. In it you will find the recipes you need to help satisfy your Inline cravings. In this section I’ll review some of the tastier morsels. Bon Appetit!

External Libraries

The most common real world need for Inline is probably using it to access existing compiled C code from Perl. This is easy to do. The secret is to write a wrapper function for each function you want to expose in Perl space. The wrapper calls the real function. It also handles how the arguments get passed in and out. Here is a short Windows example that displays a text box with a message, a caption and an ‘‘OK’’ button:

```
    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* Caption, char* Text) {
      return MessageBoxA(0, Text, Caption, 0);
    }
```

This program calls a function from the MSWin32 `user32.dll` library. The wrapper determines the type and order of arguments to be passed from Perl. Even though the real `MessageBoxA()` needs four arguments, we can expose it to Perl with only two, and we can change the order. In order to avoid namespace conflicts in C, the wrapper must have a different name. But by using the `PREFIX` option (same as the XS `PREFIX` option) we can bind it to the original name in Perl.

List Context - Using the Perl Stack

C functions can only return one value. Perl can return a list of values. How can we write a C function that returns a list of values? The answer lies in understanding how Perl really calls its subroutines. It uses an internal stack of Scalar Values or `SV*`. (Remember those from the second example?) This stack is known to the Perl experts as ‘‘the Stack’’. When Perl calls a subroutine, it puts all of your arguments on the Stack. You normally access these through the special array variable, `@_`. When the subroutine returns, Perl replaces the input parameters with the return values.

To do this from C, you need to manipulate the Stack yourself. Luckily for you, Inline provides a bunch of macros to do this easily. Here is an example that works similarly to Perl's `localtime()` function:

```
print map {"$_\n"} get_localtime(time);


use Inline C => <<'END_OF_C_CODE';
#include <time.h>
void get_localtime(int utc) {
  struct tm *ltime = localtime(&utc);
  Inline_Stack_Vars;
  Inline_Stack_Reset;
  Inline_Stack_Push(newSViv(ltime->tm_year));
  Inline_Stack_Push(newSViv(ltime->tm_mon));
  Inline_Stack_Push(newSViv(ltime->tm_mday));
  Inline_Stack_Push(newSViv(ltime->tm_hour));
  Inline_Stack_Push(newSViv(ltime->tm_min));
  Inline_Stack_Push(newSViv(ltime->tm_sec));
  Inline_Stack_Push(newSViv(ltime->tm_isdst));
  Inline_Stack_Done;
}
END_OF_C_CODE
```

Simple Perl Internals

Without knowing it you have just been introduced to Perl internals. Perl has a rich API that you can call from your C code. All of the specifics are detailed in the `perlapi` manpage. (7) Through the API you can access all of the internals of Perl as well as create new data structures. Here we present a C function that will read a file by name, parse it into words, and return a new hash of arrays containing that data. (It is the Inline version of an example from the Camel book.) The file contents look like this:

```
flintstones fred barney
jetsons     george jane elroy
simpsons    homer marge bart
```

Here is the program:

```
use Inline C;
use Data::Dumper;
$hash_ref = load_data("./cartoon.txt");
print Dumper $hash_ref;


__END__
__C__
static int next_word(char**, char*);

SV* load_data(char* file_name) {
    char buffer[100], word[100], * pos;
    AV* array;
    HV* hash = newHV();
    FILE* fh = fopen(file_name, "r");
    while (fgets(pos = buffer, sizeof(buffer), fh)) {
        if (next_word(&pos, word)) {
            hv_store(hash, word, strlen(word),
                     newRV_noinc((SV*)array = newAV()), 0);
            while (next_word(&pos, word))
                av_push(array, newSVpvf("%s", word));
        }
    }
    fclose(fh);
    return newRV_noinc((SV*) hash);
}


static int next_word(char** text_ptr, char* word) {
    char* text = *text_ptr;
    while(*text != '\0' &&
          *text <= ' ')
        text++;
    if (*text <= ' ')
        return 0;
    while(*text != '\0' &&
          *text > ' ') {
        *word++ = *text++;
    }
    *word = '\0';
    *text_ptr = text;
    return 1;
}
```

The internal calls like `newHV()` and `newRV_noinc()` may seem a bit strange, but once you read the doc, they aren't so bad. Running this program produces:

```
$VAR1 = {
          'flintstones' => [
                             'fred',
                             'barney'
                           ],
          'simpsons' => [
                          'homer',
                          'marge',
                          'bart'
                        ],
          'jetsons' => [
                         'george',
                         'jane',
                         'elroy'
                       ]
        };
```

It Takes All Types

Before version 0.30, Inline only supported five C data types. These were: `int`, `long`, `double`, `char*` and `SV*`. This was all you needed. All the basic Perl scalar types are represented by these. Fancier things like references could be handled by using the generic `SV*` (scalar value) type, and then doing the mapping code yourself, inside the C function.

The process of converting between Perl's `SV*` and C types is called **typemapping**. In XS, you normally do this by using `typemap` files. A default `typemap` file exists in every Perl installation in a file called `/usr/lib/perl5/5.6.1/ExtUtils/typemap` or something similar. This file contains conversion code for over 20 different C types, including all of the Inline defaults.

As of version 0.30, Inline no longer has *any* built in types. It gets all of its types exclusively from `typemap` files. Since it uses Perl's default `typemap` file for its own defaults, it actually has many more types available automatically.

This setup provides a lot of flexibility. You can specify your own `typemap` files through the use of the `TYPEMAPS` configuration option. This not only allows you to override the defaults with your own conversion code, but it also means that you can add new types to Inline as well. The major advantage to extending the Inline syntax this way is that there are already many typemaps available for various APIs. And if you've done your own XS coding in the past, you can use your existing `typemap` files as is. No changes are required.

Let's look at a small example of writing your own typemaps. For some reason, the C type `float` is not represented in the default Perl `typemap` file. I suppose it's because Perl's floating point numbers are always stored as type `double`, which is higher precision than `float`. But if we wanted it anyway, writing a `typemap` file to support `float` is trivial.

Here is what the file would look like:

```
    float                   T_FLOAT


    INPUT
    T_FLOAT
            $var = (float)SvNV($arg)
```

```
OUTPUT
T_FLOAT
        sv_setnv($arg, (double)$var);
```

Without going into details, this file provides two snippets of code. One for converting a `SV*` to a float, and one for the opposite. Now we can write the following script:

```
use Inline C => DATA =>
           TYPEMAPS => './typemap';


print '1.2 + 3.4 = ', fadd(1.2, 3.4), "\n";


__END__
__C__
float fadd(float x, float y) {
    return x + y;
}
```

use Inline with => Event;

`Event.pm` is a module that allows you to define callback subroutines for certain events that can happen in Perl. It also has a C interface for defining the callbacks as C functions. When Inline was first introduced to the CPAN, Jochen Stenzel quickly figured out that Event and Inline could easily be used together to define C callbacks for Event. A simple program looked like this:

```
use Event;
use Inline;
use Config;


Inline::Config::makefile(INC => "-I$Config{installsitearch}/Event");
Inline->import(C => join('', <DATA>));


Event->timer(desc     => 'Perl timer',
             interval => 0.5,
             cb       => \&c_callback,
            );
BOOT();
Event::loop;


__END__
#include "EventAPI.h"


void c_callback(SV * sv) {
  pe_event * event = GEventAPI->sv_2event(sv);
  pe_timer * watcher = event->up;
  printf("Here is the C callback (of watcher \"%s\").\n\tI detected %d events.\n
         SvPVX(watcher->base.desc),
         event->hits,
         event->prio,
         watcher->base.prio
        );
}
```

```
    void BOOT() {I_EVENT_API("Inline Script");}
```

Although this is much simpler than doing something similar in XS, it left much to be desired. For instance, the C part requires including the `EventAPI` header file. Since this is in a non-standard place, Inline needed to be informed of the correct include path. Also, the callback gets passed a pointer to a `pe_event` structure which needs to be mapped explicitly from a SV*. Finally, Event requires a bootstrap function to be called explicitly.

As of version 0.30, Inline supports a `with` syntax which informs Inline that another Perl module is Inline-enabled. That means the other module can pass hints to Inline that are hidden from the user, making the code very readable. As of version 0.80, `Event.pm` comes with Inline support. This allows us to write the previous example like this:

```
    use Inline with => Event;
    use Inline C;


    Event->timer(desc     => 'Perl timer',
                 interval => 0.5,
                 cb       => \&c_callback,
                );
    Event::loop;


    __END__
    __C__
    void c_callback(pe_event * event) {
      pe_timer * watcher = event->up;
      printf("Here is the C callback (of watcher \"%s\").\n\tI detected %d events.\n
             SvPVX(watcher->base.desc),
             event->hits,
             event->prio,
             watcher->base.prio
            );
     }
```

To implement this change, Joshua N. Pritikin needed to add only a dozen lines of code to his Event module. This is a great example of how module authors can easily expose a C interface to their users through Inline.

Calling back to Perl from C

When you use Inline to jump from the warm fuzzy pleasure palace of Perl, to the cold dark wasteland of C, you may find yourself longing for home. Don't worry my friend, you are not alone. Perl, Herself, is watching over you. Literally.

Since your C code is running under Perl, you can easily call back to Perl. The easiest way to do this is with the `eval()` command, known in C as `eval_pv()`. Here is a simple example:

```
    use Inline C;
    goto_C();

    __END__
    __C__
    void goto_C() {
        printf("I've been banished to C, but at least I have Perl %s\n",
               SvPVX(eval_pv("use Config; $Config{version}", 0)));
    }
```

Since `eval()` always returns the value of the last expression, using `eval_pv()` is probably the easiest way to execute an arbitrary Perl expression and get a scalar in return. In this example, we use `eval()` to load `Config.pm`, so we can return the Perl version number. Then we use `SvPVX()` to convert the scalar to a `char*` so that we can print it with `printf()`.

It is also possible to call back to your own Perl subroutine using functions like `call_pv()`. Here is another short example:

```
    use Inline C;
    goto_C();
    sub how_is_perl_doing {
        print "This is Perl. I'm doing fine!\n";
    }
    __END__
    __C__
    void goto_C() {
        printf("C is boring. I wonder how my friend Perl is doing?\n");
        call_pv("how_is_perl_doing", G_VOID);
    }
```

Actually calling Perl subroutines can get pretty tricky, especially when you start passing arguments back and forth. That's because you need to deal with all of the Perl Stack issues manually. For a good primer on this subject, consult the `perlcall` manpage.

Object Oriented Inline

Consider the following program:

```
    my $obj1 = Soldier->new('Benjamin', 'Private', 11111);
    my $obj2 = Soldier->new('Sanders', 'Colonel', 22222);
    my $obj3 = Soldier->new('Matt', 'Sergeant', 33333);


    for my $obj ($obj1, $obj2, $obj3) {
        print ($obj->get_serial, ") ",
               $obj->get_name, " is a ",
               $obj->get_rank, "\n");
    }

    #---------------------------------------------------------
    package Soldier;
    use Inline C => <<'END';
    typedef struct {
        char* name;
        char* rank;
        long  serial;
    } Soldier;


    SV* new(char* class, char* name, char* rank, long serial) {
        Soldier* soldier = malloc(sizeof(Soldier));
        SV*      obj_ref = newSViv(0);
        SV*      obj = newSVrv(obj_ref, class);

        soldier->name = strdup(name);
        soldier->rank = strdup(rank);
        soldier->serial = serial;


        sv_setiv(obj, (IV)soldier);
        SvREADONLY_on(obj);
        return obj_ref;
    }


    char* get_name(SV* obj) {
        return ((Soldier*)SvIV(SvRV(obj)))->name;
    }


    char* get_rank(SV* obj) {
        return ((Soldier*)SvIV(SvRV(obj)))->rank;
    }


    long get_serial(SV* obj) {
        return ((Soldier*)SvIV(SvRV(obj)))->serial;
    }

    void DESTROY(SV* obj) {
        Soldier* soldier = (Soldier*)SvIV(SvRV(obj));
        free(soldier->name);
        free(soldier->rank);
        free(soldier);
    }
    END
```

Damian Conway has given us myriad ways of implementing OOP in Perl. This is one he might not have thought of.

The interesting thing about this example is that it uses Perl for all the OO bindings while using C for the attributes and methods.

If you examine the Perl code everything looks exactly like a regular OO example. There is a `new()` method and several accessor methods. The familiar ’arrow syntax’ is used to invoke them.

In the class definition (second part) the Perl `package` statement is used to name the object class or namespace. But that’s where the similarities end and Inline takes over.

The idea is that we call a C subroutine called `new()` which returns a blessed scalar. The scalar contains a readonly integer which is a C pointer to a Soldier struct. This is our object.

The `new()` function needs to `malloc()` the memory for the struct and then copy the initial values into it using `strdup()`. This also allocates more memory (which we have to keep track of).

The accessor methods are pretty straightforward. They return the current value of their attribute.

The last method `DESTROY()` is called automatically by Perl whenever an object goes out of scope. This is where we can free all the memory used by the object.

That’s it. It’s a very simplistic example. It doesn’t show off any advanced OO features, but it is pretty cool to see how easy the implementation can be. The important Perl call is `newSVrv()` which creates a blessed scalar.

CGI with Inline.

The problem with running Inline code from a CGI script is that Inline **writes** to a build area on your disk whenever it compiles code. Most CGI scripts don’t (and shouldn’t) be able to create a directory and write into it. Here’s a simple CGI that solves the problem:

```
#!/usr/bin/perl
use CGI qw(:standard);
use Inline Config =>
           DIRECTORY => '/usr/local/apache/Inline';
print (header,
       start_html('Inline CGI Example'),
       h1(JAxH('Inline')),
       end_html
      );


use Inline C => <<END;
SV* JAxH(char* x) {
    return newSVpvf("Just Another %s Hacker", x);
}
END
```

The solution is to explicitly tell Inline which directory to use with the `'use Inline Config => DIRECTORY => ...'` line. Then you need to give write access to that directory from the web server (CGI script).

If you see this as a security hole, then there is another option. Give write access to yourself, but read-only access to the CGI script. Then run the script once by hand (from the command line). This will cause Inline to precompile the C code. That way the CGI will only need read access to the directory (to load the shared library). Just remember that whenever you change the C code, you need to precompile it again.

How Inline Works.

Inline is a simple module. That is to say, it doesn’t do anything very difficult. It just weaves together the efforts of a lot of other programs that do very difficult things. But that’s the point. Why reinvent the wheel? Or the engine? Or the Christmas tree air freshener that hangs on the rear-view mirror? Inline just puts all these essentials together into a lean, mean programming machine.

To describe how Inline works, let’s take a look at its parts.

Digest::MD5

There is one crucial trick that makes Inline.pm work. It takes an MD5 digest (or fingerprint) of your Inline source code and uses it to determine whether or not that code needs to be compiled. An MD5 fingerprint is a virtually unique 128-bit pattern that can be generated for any arbitrary piece of text. Since it is astronomically unlikely that two texts will have the same fingerprint, it is an excellent way to determine whether or not your compiled code is up to date. (8)

Every time you run your program, Inline calculates the fingerprint and compares it with that of a specified object file. If the fingerprint matches, Inline will immediately load that object and start using it. If it doesn’t match, Inline will trigger a new compile of your source code.

The net effect is that Inline compiles your code the first time that you run a script. If you change the Perl part of the script, Inline doesn’t need to recompile. If you change the Inlined source, then a compile will be triggered.

Parse::RecDescent

I came up with the inspiration for writing Inline, at Damian Conway’s presentation on Parse::RecDescent during the TPC4 conference in Monterey CA last summer. Fittingly, it is the module that Inline uses to parse C and C++. Here is a look at Inline’s C grammar:

```
c_code:   part(s) {1}

part:     comment
        | function_definition
        {
         my $function = $item[1]->[0];
         push @{$thisparser->{data}->{functions}}, $function;
        -- lines deleted --
        }
        | anything_else

comment:  m{\s* // [^\n]* \n }x
        | m{\s* /\* (?:[^*]+|\*(?!/))* \*/  ([ \t]*)? }x

function_definition:
        -- lines deleted --

anything_else: /.*/
```

Some of the grammar has been removed for brevity, but the basic idea is that Inline considers the C code to consist of 3 distinct sub-elements: ’comments’, ’function definitions’, and ’anything else’. The first and last ones are thrown away. Inline only cares about the function definitions.

XS

Even though the long term goal of Inline is to offer a reasonable replacement for XS, Inline uses XS to implement all of the bindings between Perl and C. There is nothing wrong with XS itself. Only with the burden it puts on its users. It’s hard to learn and not very Perl-like to implement.

For each bindable function definition that Parse::RecDescent finds, Inline creates an XS wrapper to call that function.

Inline provides several configuration options for C and C++ that correlate directly to XS options. See the `Inline::C` and `perlxs` manpages for more info.

ExtUtils::MakeMaker

MakeMaker is truly the hardest working module in Perl-biz. Without it Inline might not be possible, and definitely would not be so robust and cross-platform. MakeMaker is the thing that turns a Makefile.PL into a Makefile. It has dozens of options and takes a long time to learn, let alone master.

Many of the C configuration options are proxies for MakeMaker options with the same name. This allows you to use these powerful features without having to write your own Makefile.PL.

Directory Assistance

I hope that you're wondering, ''Where does Inline do all this stuff?''. Good question! Inline relies on having a special directory where it can build and install new extensions on the fly. There are several places that Inline will search for a directory called `.Inline/`. If it can't find one, it will attempt to create a new directory called `_Inline` in some well known places. You can also tell Inline which directory to use with the `DIRECTORY` config option, or the `PERL_INLINE_DIRECTORY` environment variable. The directory structure looks like this:

```
    .Inline/
       build/
         Foo/
           Foo.xs
           Makefile.PL
       config
       errors/
       lib/
         auto/
           Foo/
             Foo.so
             Foo.inline
```

build/

This is the area used to write the source files needed to build an extension. If a compilation error occurs the build subdirectory is left intact so you can debug the problem. Otherwise, it is deleted.

config

This is a very important file which contains top-level information about your Inline installation. It will be automatically generated the first time you use a `.Inline/` directory.

errors/

As a convenience, all of the files from the most recently failed build are copied to the `errors/` directory.

lib/

This directory is exactly like a local Perl installation directory. All of your compiled extension objects are installed here, so they can be found and loaded by Perl.

Foo.so

This is your object file.

Foo.inline

This is the file that contains metadata about your object. This is where the MD5 fingerprint is stored.

Gang of Four

Once Inline has taken the time to do all this nice work for you, it can finally sit down for a coffee break. All it needs to do to build and install your extension code is to invoke the same processes you would do if you wrote it all yourself. Namely:

```
perl Makefile.PL
make
make test
make install
```

Well, we don't actually do the `make test` for obvious reasons, but everything else is real.

The only difference is that the object files don't end up in Perl's site directories. Normally that takes root permission. Inline install's the objects in your local `.Inline/lib/` directory. Inline then adds that path to @INC, so Perl can find your new extensions.

DynaLoader

The final step that Inline performs is to use DynaLoader to load the compiled object and bind its public functions to Perl. Thankfully, DynaLoader performs this very platform specific task, in a cross-platform manner.

Built for Speed

Inline is optimized for fast performance. When you run an Inline program that has already compiled its source code, Inline performs the minimal amount of overhead needed to get your program fully operational and running. The only thing it needs to do that an XS program would not, is to calculate the MD5 fingerprint of your source code, and make sure that it matches the compiled version. Thankfully, the MD5 program itself is written in C, and is therefore very fast. Extra modules like Parse::RecDescent (which is quite slow to load), never come into the picture.

On the other hand, when the fingerprint does not match, the Inline module takes a vacation. He checks over the itinerary you've prepared, phones up his module buddies, sets up camp for building your extension, does his dirty work, and calls in the maid clean up the remains. All on your expense account. The premise is: ‘‘Since we only need to compile once, we might as well take the time to do it right’’.

More Information

This section was a peripheral look at how Inline works. For more specific details, read the Inline documentation, which is distributed with Inline on CPAN. Or take a peek at the source code. (It’s just Perl :-)

Some Ware Beyond the C

The primary goal of Inline is to make it easy to use other programming languages with Perl. This is not limited to C. The initial implementations of Inline only supported C, and the language support was built directly into `Inline.pm`. Since then things have changed considerably. Inline now supports multiple languages of both compiled and interpreted nature. And it keeps the implementations in an object oriented type structure, whereby each language has its own separate module, but they can inherit behavior from the base Inline module.

In this section we’ll take a quick peek at the other Inline language modules, and take a look at the API that defines how new languages should be implemented.

Inline::CPP

On my second day working at ActiveState, a young man approached me. ‘‘Hi, my name is Neil Watkiss. I just hacked your Inline module to work with C++.’’

Neil, I soon found out, was a computer science student at a local university. He was working part-time for ActiveState then, and had somehow stumbled across Inline. I was thrilled! I had wanted to pursue new languages, but didn’t know how I’d find the time. Now I was sitting 15 feet away from my answer!

Over the next couple months, Neil and I spent our spare time turning Inline into a generic environment for gluing new languages to Perl. I ripped all the C specific code out of Inline and put it into Inline::C. Neil started putting together Inline::CPP and Inline::Python. Together we came up with a new syntax that allowed multiple languages and easier configuration.

Here is an example of an Inline C++ program:

```
    use Inline ’C++’;
    my $obj1 = Soldier->new(’Benjamin’, ’Private’, 11111);
    my $obj2 = Soldier->new(’Sanders’, ’Colonel’, 22222);
    my $obj3 = Soldier->new(’Matt’, ’Sergeant’, 33333);
    for my $obj ($obj1, $obj2, $obj3) {
        print ($obj->get_serial, ") ",
               $obj->get_name, " is a ",
               $obj->get_rank, "\n");
    }


    __END__
    __C++__
    class Soldier {
      public:
        Soldier(char *name, char *rank, int serial);
        char *get_name();
        char *get_rank();
        int get_serial();
      private:
        char *name;
        char *rank;
        int serial;
    };


    Soldier::Soldier(char *name, char *rank, int serial) {
        this->name = name;
        this->rank = rank;
        this->serial = serial;
    }


    char *Soldier::get_name() {
        return name;
    }


    char *Soldier::get_rank() {
        return rank;
    }


    int Soldier::get_serial() {
        return serial;
    }
```

This example is identical to the one we saw for using object oriented Inline C. But in C++ it's a much cleaner solution.

Inline::Python

Python is a completely different kind of animal. Since Python is an interpreted language, it can't work with Inline in the traditional manner. Python gets compiled into a bytecode much like Perl, and is then executed by the Python runtime. But Inline searches for a *shared object* with an associated MD5 fingerprint. So this caused a problem for Inline. The solution was, therefore, to change the tradition.

The initial idea that Neil and I had was to have two Inline *modes*; **compiled** mode and **interpreted** mode. This was not hard to do, but what would happen if down the road we wanted to bind to something that was neither compiled nor interpreted? This modal solution did not sit well with me.

After a little brainstorming, we discovered a solution that is much cleaner. Since each module is a subclass of Inline, we simply abstracted the concepts of 'executable objects', 'building', and 'loading'. Each language implementation provides its own methods for handling all of the steps that Inline performs.

For Inline::Python the concept of an executable has been overridden to mean a `.pydat` file. A `.pydat` file contains data about a Python program. Kind of like a shared object contains data about a C program, in an abstract sense. If Inline detects that a program's Python section is not in sync with its `.pydat` file (via the MD5 fingerprint) Inline will build a new one.

To build a new `.pydat` file Inline invokes Inline::Python's `build()` method. That is also exactly what Inline does for C to build a new `.so` or `.dll` file. Whereas Inline::C uses Parse::RecDescent to parse the C code, Python uses a completely different parser. Python actually uses `python` to precompile Python. Then it stores the result in the `.pydat` file. This makes subsequent runs load much faster. Again, just like C.

Finally, Inline calls Inline::Python's `load()` routine. For C, this would be a call to DynaLoader, which would dynamically load the module and bind the appropriate C functions to Perl. For Python, `load()` has to fire up a python interpreter and then bind all the functions in the `.pydat` file to Perl subroutines. Then when the Perl program calls one of these, it magically calls the Python code.

Here is a sample program that makes uses of Inline Python:

```
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
```

This program uses a Python regex to show that ‘‘Perl rules!’’.

Since Python supports its own versions of Perl scalars, arrays, and hashes, Inline::Python can flip-flop between them easily and logically. If you pass a hash reference to python, it will turn it into a dictionary, and vice-versa. Neil even has mechanisms for calling back to Perl from Python code. See the Inline::Python docs for more info.

The Inline API

I refer to an Inline module that supports a given programming language as an **ILSM** (Inline Language Support Module). In order to write your own ILSM you need to know the Inline API. It is very simple actually. (The hard part is in implementing it :-)

For starters your ILSM must have a name beginning with `'Inline::'`. Like `Inline::Foo` for instance. Inline keeps a registry of all of the ILSMs that are installed on a system, in the user's `.Inline/` directory. This happens automatically the first time that Inline uses that directory. That way Inline doesn't need to poll the ILSM every time Inline is used. If it did, it would need to load the ILSM, and that's expensive.

The next requirement for your ILSM is for it to be a subclass of Inline by putting `Inline` into the `@ISA` array. If you want to load the Inline module as well, you should do it with:

```
    require 'Inline';
```

The statement `'use Inline;'` will not work. Note that it is probably not necessary for you to `'require Inline'` because `Inline.pm` is the module that is loading your ILSM in the first place. Remember, it is invalid for a programmer to say:

```
    use Inline::Foo <<'END_FOO';
```

They need to say `'use Inline Foo => <<'END_FOO';` instead.

The last requirement for your ILSM is to support the API which consists of these five methods:

`register()`

This method receives no arguments. It returns a reference to a hash of ILSM meta-data. `Inline` calls this routine only when it is trying to detect new ILSMs that have been installed on a given system. Here is an example of the hash ref you might return for Foo++:

```
    {
     language => 'Foo',
     aliases => ['Foo++', 'F++'],
     type => 'compiled',
     suffix => 'foodat',
    };
```

`validate()`

This routine gets passed all configuration options that were not already handled by the base `Inline` module. The options are passed as key/value pairs. It is up to you to validate each option and store its value in the `Inline` object (which is also passed in). If a particular option is invalid, you should croak with an appropriate error message.

`build()`

This method is responsible for doing the parsing and compilation of the source code. No arguments are passed in except for the `Inline::Foo` object reference. But the object contains all the pertinent information you need to perform a build. `build()` is required to create a file of a specific name. For C, this file would be the shared object.

This is the meat of your ILSM. Since it will most likely be quite complicated, it is probably best that you study an existing ILSM like `Inline::C`.

`load()`

The `load()` is called every time Inline wants to run a Foo extension. The job of this method is to load the Foo's runtime environment, and to bind all the appropriate Foo functionality to Perl subroutines. For C and other compiled languages, Inline provides an inherited method that invokes DynaLoader. For interpreted languages (or if you just want more control) you need to provide your own `load()` method.

`info()`

This method is called when the user makes use of the `INFO` shortcut. You should return a string containing a small report about the Inlined code.

Inline comes with a manpage called `Inline-API` that explains writing ILSMs in much more detail.

See Perl Run. Run Perl, Run!

Inline is a great way to write C extensions for Perl. But is there an equally simple way to embed a Perl interpreter in a C program? I pondered this question myself one day. Writing Inline functionality for C would not be my cup of tea.

The normal way to embed Perl into C involves jumping through a lot of hoops to bootstrap a perl interpreter. Too messy for one-liners. And you need to compile the C. Not very Inlinish. I had all but given up, when it suddenly struck me. What if the whole C program was just an Inline extension!

In other words, what if you could pass your C program to a perl program that could pass it to Inline. Of course, Inline would have to bind to the `main()` function, and then just call it. I could then write my own little ‘‘C interpreter’’ in Perl. And the easiest way to use an interpreter in Unix is with the `#!` syntax. If all this was possible, I could write this program:

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

and just run it from the command line. Interpreted C!

From this inspiration, and a bit of perspiration, a new programming language was born. **CPR**. ‘‘C Perl Run’’. The Perl module that gives it life is called `Inline::CPR`. The difference between this and the other ILSMs is that you never `'use Inline => CPR;'`. When you install the module it installs two special components into your Perl bin directory: `'cpr'` and `'cpr.pl'`. Together, these form the CPR interpreter.

Of course, CPR is not really its own language, in the strict sense. But you can think of it that way. CPR is just like C except that you can call out to the Perl5 API at any time, without any extra code. In fact, CPR redefines this API with its own CPR wrapper API. For instance, instead of using the `eval_pv()` function, you can use the similar `CPR_eval()` call. Here is a familiar example:

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

There are several ways to think of CPR: ‘‘a new language’’, ‘‘an easy way to embed Perl in C’’, or just ‘‘a cute hack’’. I lean towards the latter. CPR is probably a far stretch from meeting most peoples embedding needs. But at the same time its a very easy way to play around with, and perhaps redefine, the Perl5 internal API. The best compliment I’ve gotten for CPR is when my coworker Adam Turoff said, ‘‘I feel like my head has just been wrapped around a brick’’. I hope this next example makes you feel that way too:

```
#!/usr/bin/cpr
int main(void) {
    CPR_eval("use Inline (C => q{
        char* greet() {
            return \"Hello world\";
        }
    })");

    printf("%s, I’m running under Perl version %s\n",
           CPR_eval("&greet"),
           CPR_eval("use Config; $Config{version}"));
    return 0;
}
```

Using the `eval()` call this CPR program calls Perl and tells it to use Inline C to add a new function, which the CPR program subsequently calls. I think I have a headache myself. %^(

The Future of Inline

Inline version 0.30 was written specifically so that it would be easy for other people in the Perl community to contribute new language bindings for Perl. On the day of that release, I announced the birth of the Inline mailing list, inline@perl.org. (9) This is intended to be the primary forum for discussion on all Inline issues, including the proposal of new features, and the authoring of new ILSMs.

In the year 2001, I would like to see bindings for Java, Ruby, Fortran and Bash. I don’t plan on authoring all of these myself. But I may kickstart some of them, and see if anyone’s interested in taking over. If *you* have a desire to get involved with Inline development, please join the mailing list and speak up.

My primary focus at the present time, is to make the base Inline module as simple, flexible, and stable as possible. Also I want to see Inline::C become an acceptable replacement for XS; at least for most situations. Specifically, this involves:

Interactive Debugging Tools

Currently, when your Inline code doesn’t compile, Inline tells you which build directory to look in. Then you need to go poking around to figure out what happened. Future Inline versions will have a optional feature that will prompt you with a menu of files to display in a pager, whenever an error occurs.

Hiding the MD5 Keys

When Inline needs to know if an object module is in sync with the source code, it checks the MD5 fingerprint. Since the fingerprint needs to be stored somewhere relative to the file, the obvious first choice was to mangle it into the file name, like this:

```
main_C_myscript_pl_3ca433bcac47af48ef1a5479734b2ef3.so
```

where `myscript.pl` is a Perl program using Inline C in the default (`main`) namespace. While this is convenient to implement and also to guarantee uniqueness, it has a few drawbacks. For instance, if you have several objects from different builds which differ only by their MD5 keys; how do you know which one is current, and which ones are merely artifacts? Another problem is that the full path names of these files become so long that some versions of the `tar` program can't handle them.

Another method is to create a pair of files like this:

```
myscript.so
myscript.inline
```

The special `.inline` file will contain all the validation information for the object file. For the most common cases, this is the right thing to do. Support for other cases will be handled as well, but in a slightly different fashion.

MakeMaker Tools

If you wanted to write a CPAN module that used Inline C, it is currently possible to do so. But the compilation of your code will happen during the `'make test'` instead of during the `'make'` phase. This behaviour can be corrected by providing some simple Inline commands to put in your `Makefile.PL`

Precompiled Distribution

I would like to be able to say something like:

```
perl -MInline=MAKEPPM myscript.pl
```

and have Inline generate a distributable precompiled package that I could share with other users running on the same platform. The package could then be installed elsewhere, using the command:

```
ppm install ./myscript.ppd
```

without needing any compiler. This could be great for Windows Perl users who don't have the requisite compiler.

Conclusion

Using XS is just too hard. At least when you compare it to the rest of the Perl we know and love. Inline takes advantage of the existing frameworks for combining Perl and C and packages it all up into one easy to swallow pill. As an added bonus, it provides a great framework for binding other programming languages to Perl. You might say, ''It's a 'Perl-fect' solution!''

FOOTNOTES

1. ''DWIMity'' is the attribute of ''Doing What I Mean''.

2. SWIG and XS can be considered interface definition languages for extending Perl with C and C++.

3. Adapted from a one-liner by Abigail. Abigail is by far the most prolific writer of one-liners and JAPHs.

4. Adapted from a one-liner by Gisle Aas.

5. http://www.ActiveState.com

6. http://www.cygwin.com

7. The `perlapi` manpage is only available with Perl 5.6.0 and higher, but it applies equally well to Perl 5.005. You can find it online at http://www.perldoc.com/perl5.6/pod/perlapi.html.

8. The `cons` program is a replacement for `make` that also uses MD5 instead of date/time stamps. Interestingly, it is written in Perl.

9. To subscribe to the Inline mailing list, send an email message to inline-subscribe@perl.org.

About the Author

Brian Ingerson lives in Vancouver BC, Canada. He loves programming, good beer, and his wife Jen. He currently works for ActiveState, where he has the undeniably enviable position of getting paid to help improve Perl.