```
            f(obj2);
        }
    }
```

You can use this technique to give a group of classes a particular property that can be distinguished at runtime.

Interfaces are quite useful in specifying required behavior, and they enable you to program in terms of high-level types without having to get into implementation details.

practical perl

by Adam Turoff

Adam is a consultant who specializes in using Perl to manage big data. He is a long-time Perl Monger, a technical editor for *The Perl Review*, and a frequent presenter at Perl conferences.

ziggy@panix.com

Integration with Inline

Perl is a great language, but there are some things that are best left to a compiled language like C. This month, we take a look at the Inline module, which eases the process of integrating compiled C code into Perl programs.

Perl exists to make easy things easy and hard things possible. If you are writing programs using nothing but Perl, thorny issues like memory management just go away. You can structure your program into a series of reusable Perl modules and reuse some of the many packages available from CPAN. Things start to break down if you need to use a C library that does not yet have a Perl interface. Things get hard when you need to optimize a Perl sub by converting it to compiled C code.

Languages like Perl, Java, and C# focus on helping you program *within* a managed runtime environment. Reusing C libraries cannot be done entirely within these environments. Each of these platforms offers an "escape hatch" for those rare occasions when compiled code is necessary. In Java, the Java Native Interface (JNI) serves this purpose. In C#/.Net, programs can be linked to "unmanaged code," compiled libraries that live outside the .Net environment. In Perl, integration with external libraries is performed using XSubs and the esoteric XS mini-language.

Linking compiled code into Perl, Java, or .Net is necessary for a minority of projects. It is one of those "hard things that should be possible." Compared to Perl Java and .Net, Perl's XS interface is the oldest and admittedly the least easy to use. XS is a mix of C, C macros, and Perl API functions that are preprocessed to generate a C program. The resulting C source is then compiled to produce a shared object file that is dynamically linked into Perl on demand, where it provides some Perl-to-C interface glue and access to other compiled code, such as a library to manipulate images, an XML parser, or a relational database client library.

Creating an XS program is a little tricky. The mini-language itself is documented in the perlxs manual page and the perlxstut tutorial that come with Perl. XS programs may need to call Perl API functions, which are documented in the perlguts and perlapi manual pages. You can find more information in books like *Programming Perl*, *Writing Perl Modules for CPAN*, and *Extending and Embedding Perl*.

To create simple XS wrappers around compiled libraries, start by preprocessing a C header file with the h2xs tool and write additional XS wrapper functions as necessary. Another common approach uses swig to create the necessary wrapper code without using XS explicitly. If you are knowledgeable about Perl internals, you might avoid both approaches and write XS interface code from scratch.

Writing XS wrappers is tricky, and the skill is difficult to learn. Many long-time Perl programmers avoid XS because of its complexity. The state of XS is one of the factors that led the Perl development team to start the Perl 6 project. One of the goals behind Perl 6 is the creation of a new runtime engine, Parrot, that provides a substantially simplified interface for integrating with compiled code.

Enter Inline::C

One day at the Perl Conference in 2000 (shortly after the Perl 6 project was announced), Brian Ingerson had an epiphany. Linking Perl to compiled C programs is one of those "hard things

that should be possible," but there was no good reason why it needed to be hard. He set out to create a much simpler way to integrate Perl and C in the same program.

The result is his Inline::C module, which greatly simplifies integrating Perl with compiled C code. The goal behind Inline::C is to hide *all* of the complexity of Perl/C integration behind a simple, easy-to-use interface. With Inline, this task is as simple as can be, even simpler than linking C code with Java or C# programs.

Beyond just integrating C and Perl code in the same program, Inline is about combining Perl with any number of languages in one program, using the same simple interface. Inline::C, the first and oldest Inline module, integrates C code with Perl; other Inline modules enable you to integrate Perl with C++, Java, Python, and Tcl. Some language-integration modules, like Inline::Java, are in the early stages of development, while others, like Inline::C, are more heavily used and quite stable. In fact, Inline::C is so stable that it is no longer necessary to write XS glue code to load a C library into a Perl program.

Using Inline::C

One common use of Inline::C is to embed, or "inline," C functions within a Perl program. Here is a Perl program implemented with a mix of Perl and C code:

```
#!/usr/bin/perl -w
use strict;
use Inline C => <<END_OF_C;
int add(int x, int y) {
    return x+y;
}
END_OF_C

print add(3, 4), "\n";  ## prints 7
```

The use Inline declaration takes a few parameters. The first is the string "C", which indicates that the code segment that follows (in this case, a heredoc) is a C program. The next parameter is the actual text of a C program, a simple function that adds two integers.

Later, in the Perl portion of the program, the subroutine call add(3, 4) will be handled by the C function add found earlier in this program. When this Perl script is run, the C program will be extracted, compiled, and dynamically loaded.

A Perl program can have multiple instances of inlined C code. For example:

```
#!/usr/bin/perl -w
use strict;
use Inline C => <<END_OF_C;
int add(int x, int y) {
    return x+y;
}
END_OF_C

use Inline C => <<END_OF_C;
int mult(int x, int y) {
    return x*y;
}
END_OF_C

print add(3, 4), "\n";    ## prints 7
print mult(3, 4), "\n";   ## prints 12
```

Or, more conventionally, a segment of inlined C code can contain multiple definitions:

```
#!/usr/bin/perl -w
use strict;
use Inline C => <<END_OF_C;
int add(int x, int y) {
    return x+y;
}

int mult(int x, int y) {
    return x*y;
}
END_OF_C

print add(3, 4), "\n";
print mult(3, 4), "\n";
```

However, this usage gets to be cumbersome. A more readable option is to keep the Perl parts and the C parts of a program separated. In this next example, the C portion of a program is inlined in the __DATA__ section of a Perl script. The use Inline C => "DATA"; declaration tells the Inline module to look for C code in the data segment of the current Perl program. Since the Inline module can integrate languages other than C, the __C__ token is necessary to declare that the code that follows is in C. Other material, such as Pod documentation, could precede the __C__ token and still be visible within the __DATA__ section.

```
#!/usr/bin/perl -w
use strict;
use Inline C => "DATA";

print add(3, 4), "\n";
print mult(3, 4), "\n";
__DATA__
__C__
int add(int x, int y) {
    return x+y;
}

int mult(int x, int y) {
    return x*y;
}
```

Another option is to store the C source code in another file. The Inline module prefers that source code found in external files be located in another directory. In this example, the two C functions above, add and mult, are stored in src/add_mult.c. Here is the updated Perl program:

```
#!/usr/bin/perl -w
use strict;
use Inline C => "src/add_mult.c";

print add(3, 4), "\n";
print mult(3, 4), "\n";
```

These are a few of the more common ways to integrate Perl and C in a single program. The Inline module supports other mechanisms, such as compiling and loading C code that's created at runtime. Although I can think of many reasons why I want to dynamically create *Perl* code at runtime, I can't think of a reason why I would want to dynamically create and load *C* code at runtime. Nevertheless, that option exists.

How Inline Works

When mixing C and Perl code in the same program, the C sources must be compiled before they can be used. Inline is not a C interpreter or a C compiler; rather, it is an environment for integrating pieces of a program written in Perl with other languages.

Compiling the C sources as they appear in the inlined code segments is insufficient, since wrapper code is still necessary to manage the interface between Perl and C. The process is complicated but mechanical. The Inline::C module performs all of the work that would normally be done by hand when writing XS interfaces using h2xs or swig.

When these Perl programs are first run, Inline automatically generates the necessary XS wrappers for the C functions add and mult, pre-processes that XS code into C, compiles the resulting C code, and loads the object file into the current Perl process. These object files are saved in a cache directory (usually named _Inline in the current directory), where they can be reused the next time the program is run. Programs that use Inline in this manner are a little slow to run the first time, but every time thereafter, the object files are loaded in as is, and the program runs with no noticeable overhead.

Programs tend to change over time. Each time a Perl program is run, it is read by the Perl interpreter, compiled, and run. The inlined C portion of these programs could also be modified, and running a compiled version of an out-of-date C program isn't very useful. That is why Inline uses a checksum to match up the C source and object files. If the checksums match, the compiled version is loaded immediately. If the fingerprints do not match, Inline transparently compiles the updated source code before loading it.

Advanced Uses for Inline::C

The C functions add and mult are admittedly quite simplistic. However, they show that simple C functions can be integrated into Perl programs with little effort. Inline::C handles all of the complexity of converting Perl data structures to and from simple C data types (int, long, double, and char *). Inline::C also supports passing Perl scalar variables (SV * structures in C) to and from C functions.

Long-time Perl programmers also expect to have the ability to pass a list of values to a sub and to receive one back. These techniques are also supported, but are slightly more difficult to write. Inline::C manages some of this complexity but cannot hide all of it. See the Inline::C and Inline::C-Cookbook manual pages for more details on using inlined C code with these behaviors.

Many C libraries don't deal with simple C data types, but focus on application-specific data structures. Writing interface code to create C structs, examine struct members, or operate on structs is slightly more difficult. Inline still manages to hide much of the complexity for these situations. Writing glue code to use these kinds of C libraries may require using some Perl API functions. Thankfully, examples can be found with the Inline manual pages and in the perlguts and perlapi manual pages.

While the easiest way to use Inline is to combine bits of Perl and C in the same source file, the most interesting use is to provide access to an existing C library. Perl provides built-in functions for standard trigonometric functions like sin and cos, but not for tan, asin, acos, atan, or any of their hyperbolic counterparts. All of these are provided by the standard math library, Libm. Here is a small Perl program that uses Inline to provide the necessary interfaces to these trigonometric functions:

```
#!/usr/bin/perl -w
use strict;
use Inline C => "DATA",
         ENABLE => "AUTOWRAP",
         LIBS => "-lm";

my $pi = 4*atan(1);
print "pi = $pi\n";

__DATA__
__C__
double tan(double x);
double asin(double x);
double acos(double x);
double atan(double x);
```

The use Inline declaration above turns on the "Autowrap" feature, which generates wrapper code for simple function prototypes. The LIBS => "-lm" declaration specifies options to pass to the compiler when creating the object file. In this case, the glue code that Inline generates is linked against the math library, Libm.

Building with Inline

The examples presented thus far use Inline in a manner that compiles C programs as necessary, at runtime. Normally, Perl modules that provide interfaces to C libraries compile the XS interface once, at build time. Modules are installed with both the Perl source and the compiled XS interfaces.

Inline can be used to compile interface wrappers at build time as well. Here is a small module that turns on these features. First, use h2xs to create the appropriate boilerplate module files. (Although h2xs started out as a tool to convert C header files into XS interfaces, common usage today does not involve profiling header files or creating XS stubs.)

```
[ziggy@cantillon  ~]$ h2xs -AXP Math::Libm
Writing Math/Libm/Libm.pm
Writing Math/Libm/Makefile.PL
Writing Math/Libm/test.pl
Writing Math/Libm/Changes
Writing Math/Libm/MANIFEST
[ziggy@cantillon ~]$
```

Next, update the newly created Perl module, Math/Libm/Libm.pm, to include the necessary Inline magic:

```
package Math::Libm;
use 5.008;
use strict;
use warnings;

use Inline C => "DATA",
        ENABLE => "AUTOWRAP",
        LIBS => "-lm",
        NAME => "Math::Libm",
        VERSION => '1.00';

our $VERSION = '1.00';

1;
__DATA__
__C__
double tan(double x);
double asin(double x);
double acos(double x);
double atan(double x);
// ... other libm prototypes ...
```

This use Inline declaration uses two new options, NAME and VERSION. These options tell Inline to build the C wrapper code as if it were a typical XS interface.

Finally, update the autogenerated Makefile.PL. The standard use ExtUtils::MakeMaker should be replaced with a use Inline::MakeMaker declaration. At this point, the standard build/test/install process will use Inline to create, build, compile, and install a Perl module that loads the compiled interface from the site library and will not compile the C code the first time the module is used.

Building this module uses the following familiar steps:

```
[ziggy@cantillon ~/Math/Libm]$ perl Makefile.PL
Checking if your kit is complete...
Looks good
Writing Makefile for Math::Libm
[ziggy@cantillon ~/Math/Libm]$ make
cp Libm.pm blib/lib/Math/Libm.pm
/usr/bin/perl -Mblib -MInline=NOISY,_INSTALL_ -
MMath::Libm -e1 1.00 blib/arch
... Inline Diagnostic messages ...
[ziggy@cantillon ~/Math/Libm]$ make test && make install
...
[ziggy@cantillon ~/Math/Libm]$
```

Conclusion

Integrating Perl with C used to be a chore. With Inline, integrating with simple C functions is easy, and integrating with more complex C functions is possible. Using Inline is much easier than the alternatives, like writing XS code from scratch or using Java or .Net interfaces to integrate C libraries.