CHAPTER 2

Extending Perl: an introduction

This chapter will introduce the fundamentals of interfacing Perl to the C programming language; we assume you have a basic understanding of C, as described in chapter 1. Before we can describe how to do this, we must first explain how Perl modules work and how they are created.

2.1 PERL MODULES

This section describes the anatomy of a Perl module distribution. If you are already familiar with how to create pure Perl modules, then you can safely skip to the next section. In essence, a Perl module is simply a file containing Perl code (usually in its own namespace, using the `package` keyword) with a file extension of `.pm`. When you `use` a module, `perl` searches through a series of directories (specified by the `@INC` array) looking for a file with the correct name. Once found, the file is parsed and the routines are made available to the main program. This mechanism allows code to be shared and re-used and is the reason behind the success of the Comprehensive Perl Archive Network (CPAN; http://www.cpan.org/).

To maximize the reusability of your modules, you should write them in such a way that they do not interfere with other parts of Perl or other modules. If you don't, your modules may clash with other modules or with the main program—and this behavior is undesirable. You can do so in three primary ways:

- You should assign a namespace to each module. This namespace is usually the same as the module name but does not have to be. As long as another part of your program does not choose the identical namespace, the module will interact with the caller only through its defined interface.
- Your modules should export subroutines by request rather than by default. If all the subroutines provided by a module are exported, then it is possible that they will clash with other subroutines already in use. Exporting subroutines by request is particularly important if you add new subroutines to a module after writing the main program, because you may add a routine that will overwrite a previous definition. Doing so is not relevant when you define object-oriented classes, because they never export subroutines explicitly.
- You should use lexical variables (those declared with `my`) in modules wherever possible to limit access from outside the namespace and to make it easier for the module to become thread-safe.[1] Globals should be used only when absolutely necessary; in many cases you can limit them to `$VERSION` for version numbering, `$DEBUG` for switching debugging state, and the `Exporter` globals (in other words, globals that are not modified during program execution).

Here is an example of a minimalist module that shows how you can implement these constraints:

```
package Example;

use 5.006;
use strict;

use base qw/Exporter/;

our $VERSION = '1.00';

our @EXPORT_OK = qw/ myfunc /;

# Code
sub myfunc { my $arg = shift; return $arg; }

1;
```

[1] We will not attempt to cover thread safety here. All you need to know for this book is that global variables and static memory hinder the use of threads, because you must make sure parallel threads do not change the information in a variable while another thread is using the value. If you only use Perl lexical variables (limited to subroutine scope rather than file scope) and C automatic variables, you will be fine.

The first line is the namespace declaration. All code in this file is visible only in this namespace unless explicitly referred to from outside or until another package statement is encountered.

The next line makes sure that the Perl version used by this module is at least version 5.6.0 (we use the old numbering style of 5.006 to ensure that older versions of Perl will be able to understand the version). This check is necessary because the module uses the `our` variable declaration, which was introduced in this version of Perl.

All Perl modules should have `strict` checking. Among other things, this pragma instructs Perl to tell you about any undeclared variables it comes across; it's an excellent way to avoid many bugs in code.

Next, we inherit methods from the `Exporter` class in order to enable exporting of subroutines and variables to the namespace that `use`s this module.

The following line defines the version number of the module. It is used by CPAN for indexing and enables Perl to check that the correct version of a module is available.

The `@EXPORT_OK` array contains a list of all the subroutines that can be exported by this routine. They will not be exported unless explicitly requested. The `@EXPORT` array can be used to always export a function, but that functionality is not desirable in most cases.

The code on the next-to-last line implements the module functionality. The actual code for the module goes here.

Finally, all modules that are read into Perl must finish with a true value (in this case `1`) so that Perl can determine whether the module was read without error.

If we name this file `Example.pm`, we can load it with

```
use Example qw/ myfunc /;
```

in order to import the named function into the current namespace. Alternatively, if we load it as

```
use Example;
```

the function `myfunc` will not be imported but can still be accessed as `Example::myfunc()`. You can find more information about Perl modules in the `perlmod` man page that comes with Perl.

2.1.1 Module distributions

With a single Perl-only module, installation could consist simply of copying the file to a location that Perl searches in, or changing the `PERL5LIB` environment variable so that it contains the relevant directory. For anything more complex, or if the module is to be distributed to other sites (for example, via CPAN), Perl provides a framework you can use to automate installation. In order to use this framework, you need to create a number of files in addition to the module (see figure 2.1).

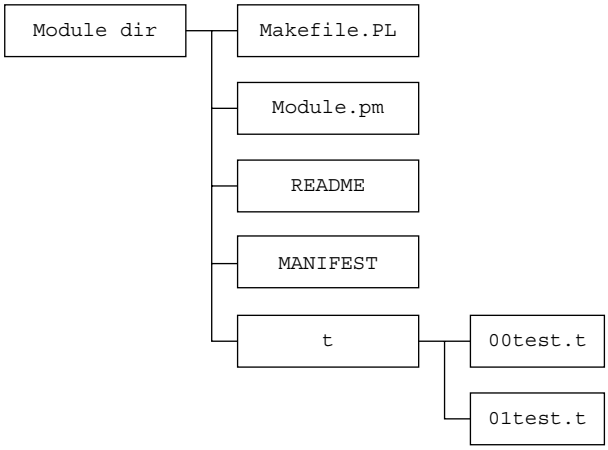

Figure 2.1
Directory tree for a simple module

The README file

This file provides a short description of the module, tells how to install it, and gives any additional information the author wants to add. The file is not required by Perl but is useful to have; it is required for any module submitted to CPAN.

The Makefile.PL file

Along with the module itself, this is the most important file that should be supplied to help build a module. It is a Perl program that generates a make file (called Makefile) when run.[2] This make file is used to build, test, and install the module. A Perl module installation usually consists of the following four lines:

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

The first line generates make file. The second line uses make file to build the module. The third line runs any included tests, and the last line installs the module into the standard location.

The Makefile.PL program is useful because it deals with all the platform-specific options required to build modules. This system guarantees that modules are built using the same parameters used to build Perl itself. This platform configuration information is installed, when Perl itself is configured and installed, as the `Config` module.

---

[2] A *make file* is a list of rules used by the make program to determine what action to take. `make` is a standard program on most Unix distributions. On Microsoft Windows and other operating systems that lack compilers as standard, you'll need to install a version of `make`.

At its simplest, Makefile.PL is a very short program that runs one subroutine:

```
use ExtUtils::MakeMaker;
# See lib/ExtUtils/MakeMaker.pm for details of how to influence
# the contents of the Makefile that is written.
WriteMakefile(
    'NAME'       => 'Example',
    'VERSION_FROM'    => 'Example.pm', # finds $VERSION
    'PREREQ_PM'       => {}, # e.g., Module::Name => 1.1
);
```

All the system-dependent functionality is provided by the `ExtUtils::MakeMaker` module. The `WriteMakefile` routine accepts a hash that controls the contents of the make file. In the previous example, `NAME` specifies the name of the module, `VERSION_FROM` indicates that the version number for the module should be read from the `VERSION` variable in the module, and `PREREQ_PM` lists the dependencies for this module (the `CPAN` module uses it to determine which other modules should be installed; in this example there are no additional module dependencies, so this isn't really required). Additional options will be described in later sections; you can find a full description in the documentation for `ExtUtils::MakeMaker`.

The MANIFEST file

This file contains a list of all files that are meant to be part of the distribution. When the Makefile.PL program is run, it checks this file to make sure all the required files are available. This file is not required, but is recommended in order to test the integrity of the distribution and to create a distribution file when using `make dist`.

The test.pl file and the t directory

Although it isn't a requirement, all module distributions should include a test suite. Test files are important for testing the integrity of the module. They can be used to ensure that the module works now, that it works after improvements are made, and that it works on platforms that may not be accessible to the module author. When `ExtUtils::MakeMaker` sees a test.pl file in the current directory, the resulting `Makefile` includes a test target that will run this file and check the results.

A good test suite uses all the functionality of the module in strange ways. A bad test suite simply loads the module and exits; a module with no test suite is even worse.

Perl provides two means of testing a module. The test.pl file is the simplest, but a more extensible approach is to create a test directory (called simply `t`) containing multiple tests. The convention is that tests in the `t` directory have a file suffix of .t and are named after the functionality they are testing (for example, loading.t or ftp.t).

Test programs are written using the framework provided by the `Test` module (or the new `Test::Simple` and `Test::More` modules). A simple example, based on this `Example` module, could be as shown in listing 2.1.

Listing 2.1 Simple test program

```
use strict;
use Test;

BEGIN { plan tests => 2 }

use Example qw/ myfunc /;
ok(1);

my $result = myfunc(1);
ok( $result, 1 );
```

The second line loads the testing framework. Then, the program informs the test framework to expect two test results.

The next line loads the module that is being tested and imports the required routines. The `ok` subroutine takes the supplied argument and checks to see whether it is `true` or `false`. In this case, the argument is always `true`, so an `ok` message is printed whenever the module has been loaded correctly.

In the last line, the `ok` routine accepts two arguments: the result from the current test and the expected result. An `ok` message is printed if the two arguments are equal; a `not ok` message is printed if they are different.

You can create templates for these files using the h2xs program that comes as part of Perl. When used with the `-X` option, it creates a basic set of files:

```
% h2xs -X Example
Writing Example/Example.pm
Writing Example/Makefile.PL
Writing Example/README
Writing Example/test.pl
Writing Example/Changes
Writing Example/MANIFEST
```

In addition to the files described here, h2xs generates a file called `Changes` that you can use to track changes made to the module during its lifetime. This information is useful for checking what has happened to the module; some editors (such as Emacs) provide an easy means of adding to these files as the code evolves.

NOTE If you want to learn more about building module distributions, we suggest you take a look at the `perlnewmod`, `perlmodlib`, and `perlmodstyle` Perl manual pages.

2.2 INTERFACING TO ANOTHER LANGUAGE: C FROM XS

Now that we have discussed how to create a module and determined that you need to create an interface to other languages, this section will describe the basics of how to combine C code with Perl. We begin with C because it is the simplest (Perl itself is written in C). Additionally, this section will only describe how to interface to C using facilities that are available in every Perl distribution. We'll describe interfacing to C using other techniques (such as SWIG and the `Inline` module) in chapter 7. If you are familiar with the basics, more advanced XS topics are covered in chapter 6.

Perl provides a system called XS (for eXternal Subroutines) that can be used to link it to other languages. XS is a glue language that is used to indicate to Perl the types of variables to be passed into functions and the variables to be returned. The XS file is translated by the XS compiler (xsubpp) into C code that the rest of the Perl internals can understand. In addition to the XS file, the compiler requires a file that knows how to deal with specific variable types (for input and output). This file is called a *typemap* and, for example, contains information about how to turn a Perl scalar variable into a C integer.

This section will begin by describing the changes that must be made to a standard Perl module in order to use XS. We'll then present an example of how to provide simple C routines to Perl.

2.2.1 The Perl module

As a first example, we will construct a Perl module that provides access to some of the examples from chapter 1. The first thing we need to do is to generate the standard module infrastructure described in section 2.1 using `h2xs`, but this time without the `-X` option to indicate that we are writing an XS extension. The module can be called `Example`:

```
% h2xs -A -n Example
Writing Example/Example.pm
Writing Example/Example.xs
Writing Example/Makefile.PL
Writing Example/README
Writing Example/test.pl
Writing Example/Changes
Writing Example/MANIFEST
```

The `-A` option indicates to h2xs that constant autoloading is not required (more on that topic later; see section 2.3.3). The `-n` option specifies the name of the module in the absence of a C header file. Besides the creation of the Example.xs file, the only difference from the previously discussed pure Perl module generation is a change to the module itself (the `.pm` file) so that it will load the compiled C code. The module created by `h2xs` has many features that are not important for this discussion, so we

will begin from the minimalist module described in section 2.1 and modify it to support shared libraries:

```
package Example;

use 5.006;
use strict;

use base qw/Exporter DynaLoader/;

our $VERSION = '0.01';
our @EXPORT_OK = qw/ print_hello /;

bootstrap Example $VERSION;
1;
```

There are only two changes (highlighted in the code). The first is that the module now inherits from the `DynaLoader` module as well as the `Exporter` module. The `DynaLoader` module provides the code necessary to load shared libraries into Perl. The shared libraries are created from the XS code on systems that support dynamic loading of shared libraries.[3] The second change is the line added just before the end of the module. The bootstrap function does all the work of loading the dynamic library with the name `Example`, making sure its version matches `$VERSION`. The bootstrap function is technically a method that is inherited from the `DynaLoader` class and is the equivalent of

```
Example->bootstrap($VERSION);
```

2.2.2 The XS file

Now that the preliminaries are taken care of, the XS file must be examined and edited. The first part of any XS file is written as if you are writing a C program, and the contents are copied to the output C file without modification. This section should always begin by including the standard Perl include files so that the Perl internal functions are available:

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

These lines must always be here and are not automatically added by the xsub compiler (although `h2xs` includes them) when it generates the C file. Any other C functions or definitions may be included in the first section of the file.

As a first example of interfacing Perl to C, we will try to extend Perl to include functions described in chapter 1. These must be added to the .xs file directly or included from either a specially built library or a separate file in the distribution. For

[3] On other systems, it is still possible to use `DynaLoader`; however, the module must be statically linked into the Perl binary by using `make perl` rather than just `make` for the second stage.

simplicity, we will begin by adding the code from sections 1.1 and 1.5.1 directly to the `.xs` file:

```
#include <stdio.h>

void print_hello (void)
{
   printf("hello, world\n");
}

int treble(int x)
{
   x *= 3;
   return x;
}
```

On the first line we have replaced the name of the `main` function from section 1.1 with a new name. XS modules do not define a `main`, because all functions are supposed to be called from somewhere else.

The XS part of the file is indicated by using the `MODULE` keyword. It declares the module namespace and defines the name of the shared library that is created. Anything after this line must be in the XS language. The name of the Perl namespace to be used for subroutines is also defined on this line, thus allowing multiple namespaces to be defined within a single module:

```
MODULE = Example  PACKAGE = Example
```

Once the module and package name have been declared, the XS functions can be added.

2.2.3 Example: "Hello, world"

As a first example, we will call the `print_hello` function declared at the start of the file. It has the advantage of being the simplest type of function to call from XS because it takes no arguments and has no return values. The XS code to call it is therefore very simple:

```
void
print_hello()
```

We begin by specifying the type of value returned to Perl from this function. In this case nothing is returned, so the value is `void`. We then give the name of the function as seen from Perl and the arguments to be passed in.

An XS function (also known as an XSUB) consists of a definition of the type of variable to be returned to Perl, the name of the function with its arguments, and then a series of optional blocks that further define the function. The `print_hello` function is very simple, so XS needs no extra information to work out how to interface Perl to the function.

IMPORTANT Unlike a C prototype, XS must have the return type (in this case, `void`) by itself on the first line of the declaration. The function name and arguments appear on the next line.

Our XS file now contains the following:

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

#include <stdio.h>

void print_hello (void)
{
   printf("hello, world\n");
}

int treble(int x)
{
   x *= 3;
   return x;
}

MODULE = Example  PACKAGE = Example

void
print_hello()
```

If we save this file as `Example.xs`, we can build the module in the *normal* way (don't forget to add this file to the `MANIFEST` if it is not there already). Listing 2.2 shows the output from Perl 5.6.0 (with some lines wrapped to fit on the page).

Listing 2.2 Output from the build of our first XS example

```
% perl Makefile.PL
Checking if your kit is complete...   ❶
Looks good
Writing Makefile for Example
% make
mkdir blib   ❷
mkdir blib/lib
mkdir blib/arch
mkdir blib/arch/auto
mkdir blib/arch/auto/Example
mkdir blib/lib/auto
mkdir blib/lib/auto/Example
cp Example.pm blib/lib/Example.pm   ❸
/usr/bin/perl -I/usr/lib/perl5/5.6.0/i386-linux
  -I/usr/lib/perl5/5.6.0
  /usr/lib/perl5/5.6.0/ExtUtils/xsubpp
  -typemap /usr/lib/perl5/5.6.0/ExtUtils/typemap Example.xs
  > Example.xsc   ❹
  && mv Example.xsc Example.c
Please specify prototyping behavior for Example.xs
```

```
(see perlxs manual)   ❺
gcc -c  -fno-strict-aliasing -O2  -DVERSION=\"0.01\"   ❻
  -DXS_VERSION=\"0.01\" -fPIC -I/usr/lib/perl5/5.6.0/i386-linux/CORE
   Example.c
Running Mkbootstrap for Example ()
chmod 644 Example.bs
LD_RUN_PATH="" gcc -o blib/arch/auto/Example/Example.so   ❼
  -shared -L/usr/local/lib Example.o
chmod 755 blib/arch/auto/Example/Example.so
cp Example.bs blib/arch/auto/Example/Example.bs
chmod 644 blib/arch/auto/Example/Example.bs
```

❶ This line checks that all the relevant parts of the distribution are present by comparing the contents of the directory with the contents listed in the `MANIFEST` file.

❷ We create the directory structure that will receive the module files as the build proceeds. This directory is called `blib` (for "build library").

❸ This line copies all the Perl files to the architecture-independent directory.

❹ We run the XS compiler, which translates the XS file to C code. The compiled file is written to a temporary file and then moved to Example.c rather than being written straight to the C file. This is done to prevent partially translated files from being mistaken for valid C code.

❺ This warning can be ignored. It informs us that we defined some XS functions without specifying a Perl prototype. You can remove this warning either by using `PROTOTYPES: DISABLE` in the XS file after the `MODULE` declaration or by specifying a prototype for each XS function by including a `PROTOTYPE:` in each definition.

❻ The C file generated by xsubpp is now compiled. The compiler and compiler options are the same as those used to compile Perl itself. The values can be retrieved from the `Config` module. Additional arguments can be specified in Makefile.PL.

❼ The final step in library creation is to combine all the object files (there can be more than one if additional code is required) and generate the shared library. Again, the process is platform dependent, and the methods are retrieved from the `Config` module.

When we run Makefile.PL, it now finds an `.xs` file in the directory and modifies the resulting `Makefile` to process that file in addition to the Perl module. The build procedure therefore adjusts to the presence of the `.xs` file. The additional steps in the procedure are illustrated in figure 2.2.

We don't yet have an explicit test program (if you started with `h2xs`, you have the outline of a test program, but it will only test whether the module will load correctly). However, we can test the newly built module from the command line to see what happens:

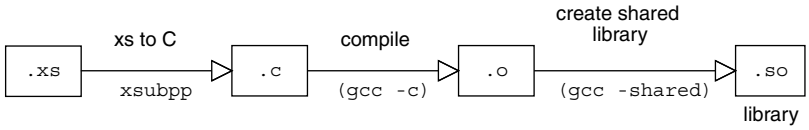

Figure 2.2 Flow diagram demonstrating the steps involved in transforming an `.xs` file to a shared library. The bracketed commands and the suffixes for object code and shared libraries will vary depending on the operating system used.

```
% perl -Mblib -MExample -e 'Example::print_hello'
Using /examples/Example/blib
hello, world
```

As expected, we now see the `"hello, world"` message. The command-line options are standard Perl but may require further explanation if you are not familiar with using Perl this way. This example uses the `-M` option to ask Perl to load the external modules `blib` and `Example` and then execute the string `Example::print_hello`. The full package name is required for the subroutine name because Perl will not import it into the `main` namespace by default. The `blib` module simply configures Perl to use a build tree to search for new modules. It is required because the `Example` module has not yet been installed.

Running tests like this is not efficient or scalable, so the next step in the creation of this module is to write a test program (or modify that generated by `h2xs`). The testing framework provided by the `Test` module[4] makes this easy. Here the test program from listing 2.1 has been modified to test our `Example` module:

```
use strict;
use Test;
BEGIN { plan tests => 2 }
use Example;

ok(1);
Example::print_hello();
ok(1);
```

If this program is saved to a file named `test.pl`, we can then use the make program to run the test:[5]

```
% make test
PERL_DL_NONLAZY=1 /usr/bin/perl -Iblib/arch -Iblib/lib
 -I/usr/lib/perl5/5.6.0/i386-linux -I/usr/lib/perl5/5.6.0 test.pl
1..1
```

[4] Prior to versions 5.6.1 of Perl, the test program created by h2xs does not use the `Test` module and is therefore more complicated than necessary.

[5] It may be necessary to rerun the make-file creation phase if the test program is created after the make file has been created. This is the case because `ExtUtils::MakeMaker` adjusts the contents of the make file depending on what is present in the module distribution.

```
ok 1
hello, world
ok 2
```

The problem with this simple test is that it is not really testing the `print_hello` subroutine but simply whether (a) the module has loaded and (b) the `print_hello` subroutine runs without crashing. Although these are useful tests, they do not tell us anything about the subroutine. This is the case because the testing system can only test variables, and the `print_hello` routine does not return anything to the caller to indicate that everything is OK. In the next section, we will fix this situation by adding a return value to the function.

2.2.4 Return values

Adding a simple return value to an XS routine (that is, a single scalar, not a list) involves telling Perl the type of return value to expect. Our `print_hello` C function does not have a return value (it returns `void`), so it must be modified. We can do so by adding the function in listing 2.3 to the top of our XS file.

Listing 2.3 "Hello, world" with a return value

```
int print_hello_retval (void)
{
   return printf("hello, world\n");
}
```

We have added a new function with a slightly different name to indicate that we are now returning an integer value. This function makes use of the fact that `printf` returns the number of characters that have been printed.

We can now add a new function to our XS code to take the return value into account:

```
int
print_hello_retval()
```

The function is identical to the XS code for `print_hello`, but the `void` declaration has been changed to an `int`. Once we've saved, we can rebuild it by typing `make` as before. If we modify the test script to add

```
my $retval = Example::print_hello_retval();
ok( $retval, 13 );
```

and change the planned number of tests to three, we can now test that the routine returns the correct value (in this case, the number of printed characters should be `13`):

```
% make test
PERL_DL_NONLAZY=1 /usr/bin/perl -Iblib/arch -Iblib/lib
-I/usr/lib/perl5/5.6.0/i386-linux -I/usr/lib/perl5/5.6.0 test.pl
```

```
1..3
ok 1
hello, world
ok 2
hello, world
ok 3
```

If the return value did not agree with the value we were expecting, the test script would have told us there was a problem:

```
% make test
PERL_DL_NONLAZY=1 /usr/bin/perl -Iblib/arch -Iblib/lib -I/usr/lib/perl5/
5.6.0/i386-linux -I/usr/lib/perl5/5.6.0 test.pl
1..3
ok 1
hello, world
ok 2
hello, world
not ok 3
# Test 3 got: '13' (test.pl at line 11)
#   Expected: '12'
```

However, it won't tell us whether there is a problem with our XS code or a bad assumption in our test script!

2.2.5 Arguments and return values

Our `treble` function from section 1.5.1 takes an integer argument and returns an integer. This function would be represented in XS as shown in listing 2.4.

Listing 2.4 XS for the treble function

```
int
treble( x )
  int x
```

The first line returns an integer. The next is the signature of the command as it will be visible to Perl; there is now an input argument. All the arguments listed in the second line are then typed in the third and successive lines. For simple C types, Perl knows the translation without having to be told.

The example from section 1.5.1 could now be written as

```
use Example;
print "Three times ten is ", Example::treble(10), "\n";
```

with the following output:

```
% perl -Mblib treble.pl
Three times ten is 30
```

2.3 XS AND C: TAKING THINGS FURTHER

So far, we have shown you how to use XS to provide wrappers to simple C functions with simple arguments where you want the signature of the Perl subroutine to match the signature of the C function. In many cases, this approach is too simplistic, and extra code must be supplied in the XS layer. The XS wrapper lets you provide C code as part of the subroutine definition using the `CODE` keyword. XS keywords occur after the initial XSUB declaration and are followed by a colon. Here are listings 2.3 and 2.4 coded entirely in XS without going through an extra function:

```
int
print_hello_retval ()
 CODE:
    RETVAL = printf("hello, world\n");
 OUTPUT:
    RETVAL

int
treble( x )
  int x
 CODE:
   RETVAL = 3*x;
 OUTPUT:
   RETVAL
```

The `CODE` keyword indicates that the following lines will contain C code. The `RETVAL` variable is created automatically by the XSUB compiler and is used to store the return value for the function; it is guaranteed to be the same type as the declared return type of the XSUB (integer in both these examples). One complication is that `RETVAL` is not automatically configured as a return value; xsubpp needs to be told explicitly that it should be returned, and this is done with the help of the `OUTPUT` keyword.

2.3.1 Modifying input variables

In some cases, input arguments are modified rather than (or as well as) providing a return value. In that case, XS needs to be told which arguments are solely for input and which are for output. You use the `OUTPUT` keyword for this purpose. Here we modify the `treble` function so that the argument is modified instead of providing the result as a return value:

```
void
treble_inplace( x )
  int x
 CODE:
  x *=3;
 OUTPUT:
  x
```

This code is equivalent to the following Perl subroutine:

```
sub treble_inplace {
  $_[0] *= 3;
  return;
}
```

Or, more pedantically:

```
sub treble_inplace {
  my $x = int($_[0]);
  $x *= 3;
  $_[0] = int($x);
  return;
}
```

It suffers from the same problem: the input argument must be a variable, not a constant, in order to be modified. If a constant is passed in (for example, a straight number, as in our previous example), Perl will generate a "Modification of a read-only value attempted" runtime error. The `OUTPUT` keyword forces the value of the variable at the end of the XSUB to be copied back into the Perl variable that was passed in.

2.3.2 Output arguments

In many C functions, some arguments are only returned (that is, the value of the argument on entry is irrelevant and is set by the function itself). In these cases, the XSUB must specify not only which arguments in the list are to be returned but which are to be ignored on input.

NOTE To be pedantic, all arguments in C are passed in by value; however, some arguments are thought of as return values because they are passed in a pointer to some memory and that memory is modified by the function. The pointer itself is not affected. Here we will use a non-pointer XS example, because XS can be used to copy the results into the correct variable. More detailed examples explicitly involving pointers can be found in chapter 6.

For example, if we wanted our `treble` function to return the result into a second argument

```
&treble(5, $out);
```

we would have to write XS code like this:

```
void
treble(in, out)
  int in
  int out = NO_INIT
 CODE:
  out = 3 * in;
 OUTPUT:
  out
```

The NO_INIT flag tells the XS compiler that we don't care what the value of the second argument is when the function is called—only that the result is stored in it when we leave. This code is functionally equivalent to the following Perl code:

```
sub treble {
  $_[1] = 3 * $_[0];
  return;
}
```

Of course, this approach preserves a C-style calling signature and forces it onto Perl. In some cases, this calling signature is desirable (maybe for familiarity with existing library interfaces), but in other cases it isn't. This brings us to the question of interface design, which is addressed in section 2.5.

2.3.3 Compiler constants

Providing access to functions is only part of the problem when interfacing to external libraries. Many libraries define constants (usually in the form of preprocessor defines) that are useful to Perl programmers as well as C programmers. h2xs automatically provides the code necessary to import preprocessor constants, unless it is invoked with the -c or -A option. The approach taken by h2xs uses the AutoLoader module to determine the value of constants on demand at runtime rather than import every constant when the program starts.

NOTE An extreme example is the standard POSIX module. It defines more than 350 compiler constants, and creating this many subroutines during loading would impose a large overhead.

The autoloading is implemented in two parts. First, an AUTOLOAD subroutine is added to the .pm file. For versions of Perl before 5.8.0, the code will look something like this:

```
use strict;
use Errno;
use AutoLoader;
use Carp;

sub AUTOLOAD {
    my $sub = $AUTOLOAD;
    (my $constname = $sub) =~ s/.*::/;
    my $val = constant($constname);
    if ($! != 0) {
        if ($! =~ /Invalid/ || $!{EINVAL}) {
            $AutoLoader::AUTOLOAD = $sub;
            goto &AutoLoader::AUTOLOAD;
        } else {
            croak "Your vendor has not defined constant $constname";
        }
    }
    {
```

Checks for explicit values of the errno variable

Loads the Carp module, which imports the croak function

```
        no strict 'refs';                    ❾
        *$sub = sub () { $val };
    }
    goto &$sub;    ❿
}
```

❶ This line loads the `AutoLoader` module. It is required only if we want to dynamically load additional functions from their corresponding files.

❷ The subroutine must be called `AUTOLOAD` so that Perl will call it automatically when it cannot find a definition for the subroutine in this package.

❸ The `$AUTOLOAD` package variable contains the name of the subroutine that is being requested. Here we copy that value to a lexical variable for convenience.

❹ This line strips the package name (and associated colons) from the requested function name, leaving just the name of the function in the current namespace.

❺ Next, we run the `constant` function that returns the value of the required constant. This routine is an XS function that is created by the h2xs command (more on that later). The `AUTOLOAD` code generated by h2xs passes a second argument to this routine (`$_[0]`), but for simple constants it can usually be removed from the routine.

❻ This line checks the error status from the `constant` function. In C, a common way of setting status is for the function to set a global variable `errno` and for the caller of the function to check `errno` when control is returned to it. In Perl, this behavior is implemented by the `$!` variable. `$!` is tied to the C `errno` variable so that Perl can check the value after system calls. Here, the `constant` function sets `errno` if the requested constant cannot be located.

❼ We now check to see if `errno` is set to a value that indicates the constant does not exist. In that case, control passes to `AutoLoader`, so it can check whether the required subroutines are to be autoloaded from .al files. These checks are required only if autoloaded routines are expected; otherwise this is an extra overhead for the program.

❽ If the constant should be available but was not defined, this line stops the program. The `croak` function is used rather than `die` so that the line number in the caller's code is printed in the error message rather than the line number in the `AUTOLOAD` subroutine.

❾ At this point in the routine, the value of the constant has been determined and, in principle, could be returned to the caller. Although it is valid to do so, in practice the constant will probably be called more than once. If the value is returned immediately, then the `AUTOLOAD` subroutine will be called every time the constant is requested—very inefficient. To overcome this inefficiency, the `AUTOLOAD` subroutine creates a

new subroutine in this package that simply returns the constant value. In the example, this is done by creating an anonymous subroutine and storing it in a glob (see section 4.6 for more details of how this process works). The name of the glob is stored in `$sub` and therefore requires that soft references be allowed; the `no strict 'refs'` turns off strict checking to allow them. If you are uncomfortable with glob assignments, you can achieve the same effect by using a string `eval`:

```
eval "sub $sub () { $val }";
```

❿ Finally, Perl is instructed to jump to the newly created subroutine and resume execution there. Using `goto` allows the program to run as if `AUTOLOAD` were never called.

The second part of the solution generated by h2xs lies in the `constant` function in the `.xs` file. Here is a simple yet functional form of the code generated by h2xs for some of the file constants that are available from the `Fcntl` module:

```
static IV                                    ❶
constant(char *name)
{
  errno = 0;                                 Sets to "no error";
                                             value checked on exit
                                             from function
  switch (*name) {                           ❷
  case 'S':                                  Executes if the constant
                                             name begins with S
     if (strEQ(name, "S_ISGID"))             Compares the
#ifdef S_ISGID                               requested name with
                                             the string S_ISGID
         return S_ISGID                      ❸
#else
         goto not_there;
#endif
         break;
  case 'O':
     if (strEQ(name, "O_RDONLY"))
#ifdef O_RDONLY
         return O_RDONLY;
#else
         goto not_there;
#endif
     if (strEQ(name, "O_RDWR"))
#ifdef O_RDWR
         return O_RDWR;
#else
         goto not_there;
#endif
     break;                                  ❹
    }
    errno = EINVAL;                          ❺
    return 0;

not_there:                                   ❻
    errno = ENOENT;
    return 0;
```

```
}

MODULE = Fcntl_demo PACKAGE = Fcntl_demo   <-- Defines the start of the XS part of the file

IV   (7)
constant(name)
   char * name
```

(1) This line indicates the return type of the function. In this case, the return type is forced to be the Perl integer type (see section 1.6.2).

(2) This line denotes the start of a block that will switch on the first character of the requested constant name.

(3) This block does all the work for the `S_ISGID` constant. The C preprocessor is used to determine the code that is passed to the compiler. If the symbol is defined, its value is returned; if it is not defined, the code branches to the `not_there` label.

(4) If the constant cannot be found even though it started with the letter *O*, the switch is exited; it isn't possible for any of the remaining case statements to match.

(5) If the constant name did not match anything in the switch block, `errno` is set to `EINVAL` ("Invalid argument") and the function returns `0`.

(6) If the requested name was valid and present in the switch but was not available (maybe because the constant was not defined on this operating system), the function sets `errno` to `ENOENT` (literally, "No such file or directory") and returns.

(7) The XSUB definition for `constant` is simple; it has a single string argument and a return type of `IV` (integer value).

It is important to realize that this example only deals with numeric constants (the constants are assumed to be integers). String constants must be handled differently—especially if a mixture of numeric and string constants is required.

ExtUtils::Constant

A new `ExtUtils::Constant` module has been added from Perl 5.8.0; it simplifies the handling of constants. With this module, the XS and C code required to deal with the compiler constants is generated automatically when Makefile.PL is executed. This approach has a number of advantages over the current scheme:

- You can make improvements in the constant-handling code without having to touch every module that uses constants.
- The XS files are much simpler. Files are no longer dominated by long, repetitive lists of constants and C preprocessor directives.
- The new system allows compiler constants to have different types. An integer constant is treated differently than a floating-point constant.

2.4 WHAT ABOUT MAKEFILE.PL?

So far, we have not addressed the contents of the file that is instrumental in configuring the build process. When you're building simple Perl modules, Makefile.PL is almost empty; it just provides the name of the module and a means for determining the location to install the module (see section 2.1.1). The Makefile.PL program is much more important when you're building XS extensions, because the make file that is generated must include information about how to translate the XS code to C, how to run the C compiler, and how to generate shared libraries. In all the examples presented so far, this process has been handled automatically by the `WriteMakefile` function because it detects the presence of an XS file and sets up the appropriate make file targets. However, this detection works only if the module can be built without additional configurations above and beyond those used to build Perl originally.

So far, the examples have not required anything more than standard include files and libraries. What happens if you build a wrapper around a library that is not included by default? Let's add the following code to the XS example to find out. It will print out the version of the XPM library on our system. The `include` directive goes after the Perl includes and the XS declaration in the XS section. A minimum XS file looks something like this:

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

#include <X11/xpm.h>

MODULE = Example PACKAGE = Example

int
XpmLibraryVersion()
```

If we add this XPM code to our example file and build, we get the following:

```
% perl Makefile.PL
Checking if your kit is complete...
Looks good
Writing Makefile for Example
% make
mkdir blib
mkdir blib/lib
mkdir blib/arch
mkdir blib/arch/auto
mkdir blib/arch/auto/Example
mkdir blib/lib/auto
mkdir blib/lib/auto/Example
cp Example.pm blib/lib/Example.pm
/usr/bin/perl -I/usr/lib/perl5/5.6.0/i386-linux
    -I/usr/lib/perl5/5.6.0
    /usr/lib/perl5/5.6.0/ExtUtils/xsubpp
    -typemap /usr/lib/perl5/5.6.0/ExtUtils/typemap Example.xs
```

```
    > Example.xsc && mv Example.xsc Example.c
Please specify prototyping behavior for Example.xs
(see perlxs manual)
gcc -c  -fno-strict-aliasing -O2 -march=i386
    -mcpu=i686 -DVERSION=\"0.01\"
    -DXS_VERSION=\"0.01\" -fPIC
    -I/usr/lib/perl5/5.6.0/i386-linux/CORE  Example.c
Running Mkbootstrap for Example ()
chmod 644 Example.bs
LD_RUN_PATH="" gcc -o blib/arch/auto/Example/Example.so  -shared
    -L/usr/local/lib Example.o
chmod 755 blib/arch/auto/Example/Example.so
cp Example.bs blib/arch/auto/Example/Example.bs
chmod 644 blib/arch/auto/Example/Example.bs
```

It looks like everything worked fine. Let's try it:

```
% perl -Mblib -MExample -e 'Example::print_hello'
Using ..../Example/blib
hello, world

% perl -Mblib -MExample -e 'print Example::XpmLibraryVersion'
Using ..../Example/blib
perl: error while loading shared libraries:
    /path/to/library/Example/blib/arch/auto/Example/Example.so:
    undefined symbol: XpmLibraryVersion
```

The output indicates that the earlier routines (such as `print_hello`) still work, but the new routine doesn't. The error message says that Perl could not find `XpmLibraryVersion` in any of the libraries it has already loaded. This is not surprising, because Perl is not linked against graphics libraries during a standard build. To overcome this problem, we can use Makefile.PL to provide the information necessary to locate the correct libraries. The Makefile.PL file generated by `h2xs` looks something like this:

```
use ExtUtils::MakeMaker;
# See lib/ExtUtils/MakeMaker.pm for details of how to influence
# the contents of the Makefile that is written.
WriteMakefile(
    'NAME'         => 'Example',
    'VERSION_FROM' => 'Example.pm', # finds $VERSION
    'PREREQ_PM'    => {}, # e.g., Module::Name => 1.1
    'LIBS'         => [' '], # e.g., '-lm'
    'DEFINE'       => ' ', # e.g., '-DHAVE_SOMETHING'
    'INC'          => ' ', # e.g., '-I/usr/include/other'
);
```

The hash provided to `WriteMakefile` can contain many different keys, but the ones that are usually modified for simple XS projects are `LIBS` and `INC`. You can use the `LIBS` key to specify additional libraries that are needed to build the module. The string must be in the form expected by the linker on your system. Usually this means

a format of `-L/dir/path -lmylib`, where `-L` indicates additional search directories and `-l` indicates the name of actual libraries.[6] `WriteMakefile` expects the `LIBS` argument to be either a simple scalar or a reference to an array. In most cases, a scalar is all that is required; but the array allows multiple sets of library combinations to be provided, and `MakeMaker` will use the first that refers to a library that can be found on disk.

In order to fix our example, we must change the `LIBS` entry so that the `Xpm` library (and associated `X11` library) will be included:

```
'LIBS'  => '-L/usr/X11R6/lib -lX11 -lXpm',
```

Rebuilding the module now gives the following:

```
% perl Makefile.PL
Writing Makefile for Example
% make
gcc -c  -fno-strict-aliasing -O2 -march=i386 -mcpu=i686
    -DVERSION=\"0.01\"
    -DXS_VERSION=\"0.01\" -fPIC
    -I/usr/lib/perl5/5.6.0/i386-linux/CORE
    Example.c
Running Mkbootstrap for Example ()
chmod 644 Example.bs
LD_RUN_PATH="/usr/X11R6/lib" gcc
    -o blib/arch/auto/Example/Example.so
    -shared -L/usr/local/lib Example.o
    -L/usr/X11R6/lib -lX11 -lXpm
chmod 755 blib/arch/auto/Example/Example.so
cp Example.bs blib/arch/auto/Example/Example.bs
chmod 644 blib/arch/auto/Example/Example.bs
```

The value specified for `LIBS` is highlighted.

The test runs as expected:

```
% perl -Mblib -MExample -e 'print Example::XpmLibraryVersion'
Using ..../Example/blib
30411
```

Similarly, you can add extra include paths using the `INC` key if you are using include files that are not in the standard locations. This value is always a scalar and contains a list of directories to search for include files, in the format expected by your compiler. This list is usually of the form

```
INC => '-I/some/dir -I/some/other/dir'
```

[6] On Unix systems, `-lmylib` refers to a file on disk called `libmylib.a` or `libmylib.so`. The former is a static library, and the latter is a shared library that is loaded at runtime.

2.4.1 It really is a Perl program

It is important to remember that Makefile.PL is a normal Perl program. All that matters is that `WriteMakefile` is called with the correct arguments to generate the make file. You can write arbitrarily complex code to generate those arguments, you can prompt the user for information (as, for example, the Makefile.PL file for the `libnet` package does), or you can even dynamically generate the Perl module itself!

As an example, suppose we wanted to build an interface to a Gnome library.[7] Most Gnome libraries come with configuration scripts that can be used to determine the required libraries and include directories, and you must use these in `Makefile.PL` rather than hard-wiring the location of the Gnome system into the program.[8] To support doing this, the Makefile.PL file may look something like this:

```
use ExtUtils::MakeMaker;

# Use gnome-config to determine libs
my $libs = qx/ gnome-config --libs gnome /;

# Use gnome-config to determine include path
my $incs = qx/ gnome-config --cflags gnome /;

# Remove newlines
chomp($libs);
chomp($incs);

# Might want to exit with an error if the $libs or $incs
# variables are empty

# See lib/ExtUtils/MakeMaker.pm for details of how to influence
# the contents of the Makefile that is written.
WriteMakefile(
    'NAME'         => 'Gnome',
    'VERSION_FROM' => 'Gnome.pm', # finds $VERSION
    'PREREQ_PM'    => {}, # e.g., Module::Name => 1.1
    'LIBS'         => $libs, # all X11 programs require -lX11
    'DEFINE'       => ' ', # e.g., '-DHAVE_SOMETHING'
    'INC'          => $incs, # e.g., '-I/usr/include/other'
);
```

2.5 INTERFACE DESIGN: PART 1

Now that you have seen how to create Perl interfaces to simple C functions and library routines, this section will provide some advice about how these C routines should behave in a Perl world. When you're interfacing Perl to another language, it is important to take a step back and design the Perl interface so that a Perl programmer would be comfortable with it rather than a C programmer.

---

[7] Modules for many Gnome libraries are already on CPAN.

[8] Gnome is usually installed into `/usr` on Linux but `/opt/gnome` on Solaris.

2.5.1 Status and multiple return arguments

In C, although all arguments are passed by value, arguments can act as input arguments, return arguments, or both, and there is no way to distinguish this behavior from the prototype (knowing you are using a pointer cannot tell you whether the data will change).[9] In Perl, input arguments are supplied and return arguments are returned. A C function such as

```
int compute(int factor, double *result);
```

that may take an input integer, store a value into a `double` (the asterisk indicates a pointer in C; we will talk about those in chapter 3), and return an integer status is almost always better written in Perl as

```
($status, $result) = compute( $factor );
```

rather than

```
$status = compute( $factor, $result );
```

In versions of xsubpp prior to v1.9508 (the version shipped with Perl 5.6.1), the only way to return multiple arguments is to manipulate the argument stack by hand (as described in chapter 6). In newer versions of xsubpp, you can indicate that some arguments are to be returned differently using modifiers when declaring the function signature:

```
REQUIRE: 1.9508

int
compute( factor, OUTLIST result )
  int factor
  double result
```

The first line makes sure we are using a version of `xsubpp` that is new enough. The `OUTLIST` keyword indicates that the argument is a return value that should be placed on the output list. In fact, if the status is only telling us whether something worked or failed, we should consider removing it

```
$result = compute( $factor );
```

and returning `undef` if an error occurs. We'll show how to do this in section 6.9.

2.5.2 Don't supply what is already known

Do not ask the Perl programmer to provide information that Perl already has. For example, a C function might need to know the size of a buffer being passed in.

---

[9] In C, the `const` modifier can be used to indicate that a variable will not change. Unfortunately, many libraries still do not use it consistently. It is therefore almost impossible for automatic tools like xsubpp to infer that a variable is an output argument, simply because it lacks a `const` in the declaration.

Because the length of Perl strings is already known, it is redundant and error-prone to ask the programmer to provide that information explicitly.

2.5.3 Don't export everything

When interfacing to a library, do not blindly import every function into Perl. Many of the functions may be support functions needed by the C interface but irrelevant to Perl. Additionally, many of the constants may not be needed.

2.5.4 Use namespaces

Use Perl namespaces. Many C libraries use the library name as a prefix to every function (for example, many function names in the Gnome library begin with `gnome_`, and function names in the XPM library begin with `Xpm`), so use the package name to indicate that information and strip the common prefix. The `PREFIX` keyword can be used to do so:

```
MODULE = Xpm  PACKAGE = Xpm  PREFIX = Xpm

int
XpmLibraryVersion()
```

This XS segment indicates that the function should appear to Perl as `Xpm::LibraryVersion` rather than the more verbose and repetitive `Xpm::Xpm-LibraryVersion`.

2.5.5 Use double precision

If a library provides a single- and double-precision interface, consider using just the double-precision interface unless there is a major performance penalty if you do. All Perl floating-point variables are already double precision, and there is little point in converting precision when transferring data between Perl and the library. If you need to preserve the function names in Perl (but, as noted in a previous comment, it may be better to adopt a more unified interface on the Perl side), you can export both the single- and double-precision names but only use the double-precision function from the library. XS provides a way to do this using the `ALIAS` keyword. For example:

```
double
CalcDouble( arg )
   double arg
 ALIAS:
    CalcFloat = 1
 CODE:
    printf("# ix = %d\n", ix );
    RETVAL = CalcDouble( arg );
 OUTPUT:
   RETVAL
```

Here, `CalcFloat` is set up as an alias for `CalcDouble`. The `ix` variable is provided automatically and can be used to determine how the function was called. In this example, if the function is called as `CalcDouble`, `ix` will have a value of `0`; if

the function is called as `CalcFloat`, `ix` will have a value of `1`. Any integer value can be used for the `ALIAS`; there is nothing special about the use of `1` as the first alias.

2.6 FURTHER READING

More information on Perl modules and XS can be found at the following locations:

- *`ExtUtils::MakeMaker`*—This man page describes `Makefile.PL` options and `MakeMaker`.
- *Managing Projects with make (2nd ed.)*—This book by Andrew Oram and Steve Talbott (O'Reilly and Associates, Inc.; ISBN 0937175900) is an introduction to the `make` command.
- *`perlmod`, `perlmodlib`*—These are the standard Perl manual pages on module creation.
- *`perlxstut`, `perlxs`*—These man pages are the standard XS tutorial and documentation that come with Perl. They cover everything about Perl and XS but they rapidly move on to advanced topics.

2.7 SUMMARY

In this chapter, you have learned the following:

- How to build a simple Perl module
- How to extend Perl using XS, first using C functions that take arguments and then with functions that can return values
- How to import constants from C header files into Perl

Finally, we discussed some of the issues of interface design.