Using Perl and C

Tim Jenness

Simon Cozens

Using Perl and C

by Tim Jenness and Simon Cozens

Draft Edition

Table of Contents

List of Tables

List of Figures

List of Examples

Chapter 1. C for Perl Programmers

When using C and Perl together, the first thing we need to realise is that they are very different languages, requiring different styles and different thought patterns. Perl spoils programmers by doing an awful amount of the hard work for them; if you've never programmed in C before, the language can feel very barren and empty. C is close to the machine - Perl is close to the user.

That being said, Perl's syntax borrows heavily from C's, and so most of the elements of a C program should be familiar to a competent Perl programmer with a little thought and a little preparation.

1.1. Hello, World

The classic book on C programming is Kernighan and Ritchie's *The C Programming Language*, which begins with a program a little like this:

Example 1-1. "Hello world", from C

```
#include <stdio.h>

int main(int argc, char **argv)
{
  printf("hello, world\n");
  return(0);
}
```

The classic book on Perl, on the other hand, is *Programming Perl*, by Larry Wall, Tom Christiansen and Randal Schwartz. This begins with a fairly similar program:

Example 1-2. "Hello world", from Perl

```
print "Howdy, world!\n";
```

First, notice that Perl is a lot more compact: there's no waffle, no housekeeping. We want to print something, we tell Perl to print it. C, on the other hand, requires a little more support from the programmer.

Let's first look at how we compile and run the C program, before looking at how it's constructed.

1.2. The C compiler

There's only one Perl, but there are a large variety of different C compilers and implementations; these can have a graphical front-end, such as the Microsoft Visual Studio, or a simple command-line interface,

such as the Free Software Foundation's GCC. We'll talk about GCC here, since it's the most popular free compiler.

The simplest way of calling GCC is to simply give it the name of a C program to compile. GCC is particularly quiet; if all is well, it will give no output on the screen:

```
% gcc hello.c
%
```

This will produce an executable called `a.out` [1] in the current directory; if we run that:

```
% ./a.out
hello, world!
```

we get our tired and worn greeting.

Sometimes we don't want our output called `a.out`, so we can tell GCC to give it another name with the `-o` option:

```
% gcc -o hello hello.c
```

Perl encourages programmers to turn on the `-w` flag for warnings; I encourage C programmers to turn on the `-Wall` flag for all warnings:

```
% gcc -Wall -o hello hello.c
```

If you have a collection of C files that make up one program, you could list them all:

```
% gcc -Wall -o bigproject one.c two.c three.c four.c
```

However, it's more popular to use GCC to convert each one to an *object file* (extension `.o` with GCC, equivalent to `.obj` files on Windows) - an intermediate stage in compilation - and then link all the object files together; this allows you to change individual files without needing to completely recompile everything. To tell GCC to produce an object file, use the `-c` flag:

```
% gcc -Wall -o one.o -c one.c
% gcc -Wall -o two.o -c two.c
% gcc -Wall -o three.o -c three.c
% gcc -Wall -o four.o -c four.c
```

And then simply list the object files to link them all together:

```
% gcc -Wall -o bigproject one.o two.o three.o four.o
```

There are more complicated ways of building large programs, using static and dynamic libraries, but we need not go into those for the examples here.

1.3. Header Files

The first line in the C program is an `include` statement: this is similar to Perl's `require` in that it instructs the language to go and find a library file and read it in. However, where Perl's `.pm` and `.pl` library files contain real Perl code, C's `.h` files (*header* files) contain only the *promise* of code - they contain function prototypes, and just like Perl's subroutine prototypes, these allow the compiler to check our use of the functions while specifying that the real code will come later.

But what function's prototype do we need, and where will the real code from? Well, the only function we use is `printf`, in line 5 of our example. Perl has a `printf` function too, and they are almost identical. However, all of Perl's functions documented in `perlfunc` are built in to the language - C, on the other hand, has *no* built-in functions. Everything is provided by a library called the "standard C library", which is included when any C program is compiled. The `printf` function comes from a section of the standard C library called the "standard IO library", and to let the C compiler ensure that we are using it properly, we have it read the header file which contains the prototypes for the standard IO library, `stdio.h`.

> It's almost like **`use strict`** is always on - we can't use any function without telling C where it's coming from.

1.4. The `main` function

The main reason the Perl version was much smaller is because everything in C must be inside a function - if you have Perl code outside of any subroutine, it will all be executed in order. C needs all of this code in one place, the `main` function.

> What you'd call a "subroutine" in Perl gets called a "function" in C.

When your program begins, C arranges for this function to be called.

This function, too, must have a prototype, and here it is again:

```
int main(int argc, char **argv)
```

Perl's prototypes only tell you what type of data is coming into the function: **`sub main($@)`** tells us that the subroutine `main` takes a scalar and an array.

In C, we're not only told what's coming into the function, we're told what variables it should be stored in, and also what type of data the function should return. Here, we're returning an integer, and we're being given an integer called `argc`, and also something called `argv`: we'll look at exactly what **`char **`** means in the next section, but you might be able to guess that `argv` is similar to Perl's `@ARGV` - an array of the command line parameters. `argc` is actually the number of elements in the array - that is, the number of command-line parameters passed to the program. (The *arg*ument *c*ount.)

The `main` function almost always has the prototype given above: it should take two parameters which represent the command line, and return an integer value to the operating system, the *exit status*. Just as in Perl, the program may end implicitly, when there is no more code to execute, or explicitly, when the `exit` function is called. Our first example ended by `return`ing a value to the operating system, ending the `main` function. We could also, theoretically, allow execution to "fall off the end" of the `main` function:

```
#include <stdio.h>

int main(int argc, char **argv)
{
  printf("hello, world\n");
  /* No return, just falls off. */
}
```

Here you see the use of a C comment - comments are made up of matched pairs of **`/*`** and **`*/`**. Note that comments cannot be nested. This is legal:

```
 /*
    Comment out some code:
    printf("This won't print\n");

    Now carry on with the program:
 */
```

But this isn't:

```
 /*
    Comment out some code:
    printf("This won't print\n"); /* A comment */

    Now carry on with the program:
 */
```

The comment will be ended at the first **`*/`** - that is, after **`A comment`** - and the C compiler will try and compile **`Now carry on...`**.

Allowing execution to fall off the end of `main` is not recommended; it produces a warning to the effect that C expected our function to return a value, and it never did. Finally, we can also call `exit` explicitly:

```
#include <stdio.h>
```

```
int main(int argc, char **argv)
{
  printf("hello, world\n");
  exit(0);
}
```

The compiler makes a special exemption for `exit` as it knows that the function will not return.

1.5. Variables and Functions

As well as having to define our function prototypes, C forces us to define our variables, just like the **`use strict`** mode of Perl. We can define four different types of variables: function parameters, automatic variables, global variables, and static variables.

1.5.1. Function parameters

Function parameters are declared in the prototype of a function. Let's look at a very simple function:

Example 1-3. Function parameters

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

Here we've taken an integer, `x`, multiplied it by 3, and returned its new value. Note that we don't need the brackets around `return` - this is because `return` is not really a function; it's actually a keyword of the language. Unlike Perl, all functions in C require brackets.

The Perl equivalent of the above function would look like this:

```
sub treble {
    my $x = shift;
    $x *= 3;
    return $x;
}
```

And we'd call it like this:

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

Well, we don't have a direct equivalent to `print` in C, but we can use `printf`:

Example 1-4. Calling a function

```
#include <stdio.h>

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

int main(int argc, char** argv) {
    printf("Three times ten is %d\n", treble(10));
    return 0;
}
```

As we mentioned above, function prototypes must appear before they are called, so that the compiler can check the types: hence the definition of `treble` must appear before the `main` function calls it. Otherwise, C assumes that the return value will be int, and you will receive a warning from the compiler if warnings are turned on.

Function parameters act just like lexical variables in Perl (`my` variables); they have the same scope as their enclosing block (which is always the function), and they are private to that scope.

1.5.2. Automatic variables

Automatic variables have similar scoping properties to function parameters, but they can appear in any block. They are directly equivalent to Perl's lexical variables.

You declare an automatic variable by simply listing its type and its name; however, declaring automatic variables *must* happen before any statements in the block. So, you may say this:

Example 1-5. Automatic variables

```
int main (int argc, char** argv) {
    int number;
    number = 4;

    printf("Hello world\n");
    return(0);
}
```

But you may not say this:

```
int main (int argc, char** argv) {
    printf("Hello world\n");

    int number;
    number = 4;
```

```
    return(0);
}
```

You can also initialize your automatic variables when you declare them, by saying, for instance **`int number = 4;`**. You may also start a new block to declare an automatic variable:

```
int some_function(int parameter) {
    int one = 1;

    {
        int two = 2;
        printf("one = %d, two = %d\n", one, two);
    }

    /* "two" is out of scope here */
    return one;
}
```

Because of this property, it's likely that most of your temporary variables - in fact, most of the variables you will use - will tend to be automatic variables. But why are they called automatic variables? Since C knows their scope and their type at compile time, it can automatically make memory allocation for them.

Just like in Perl, if we have two lexical variables with the same name, only the most recently-declared one is in scope, "hiding" the older one:

```
int main(int argc, char** argv) {
    int number = 10;
    {
        int number = 20;
        printf("%d", number); /* Will print "20" */
    }
    return number;
}
```

However, unlike Perl, C does not give us a warning in this case.

1.5.3. Global variables

If a variable is declared outside of a function, it's available to all the functions in that file:

Example 1-6. Global Variables

```
#include <stdio.h>
int counter = 0;

void bump_it() {
```

```
    counter++;
}

int main(int argc, char** argv) {
    printf("The value of counter is %d\n", counter);
    bump_it();
    bump_it();
    printf("The value of counter is %d\n", counter);
    bump_it();
    printf("The value of counter is %d\n", counter);

    return(0);
}
```

The function `bump_it` modifies the global variable `counter`, which `main` reads. `bump_it` is declared to return type void - this just means “it will not return anything”; think of “void context” in Perl.

It’s also possible to share a global variable across multiple files, by declaring it once in one file, and then prefixing the declaration in other files with the **`extern`** keyword. For instance, if we have a file called `count1.c` containing the declaration and `main` function from above:

```
#include <stdio.h>
int counter = 0;

void bump_it();

int main(int argc, char** argv) {
    printf("The value of counter is %d\n", counter);
    bump_it();
    bump_it();
    printf("The value of counter is %d\n", counter);

    return(0);
}
```

(We still need to provide the prototype to `bump_it`, just as `stdio.h` provides the prototype for `printf`.) and also a file called `count2.c` containing the `bump_it` function:

```
extern int counter;

void bump_it() {
    counter++;
}
```

We can now compile these files into object files and link them together, like this:

```
% gcc -Wall -o count1.o -c count1.c
% gcc -Wall -o count2.o -c count2.c
% gcc -Wall -o count count1.o count2.o
```

The function in `count2.c` knows that it should be able to access a global variable called `counter`, which is declared “externally” - somewhere else in the program. C finds the global declaration of `counter` in `count1.c`, and so the function can access the global variable.

1.5.4. Static Variables

The final type of variables on offer are static variables. Perl actually does offer static variables, but in an obscure, undocumented and unsupported way. What would you expect this piece of Perl code to do?

```
sub foo {
    my $x if 0;
    return ++$x;
}

for (1..10) { print foo(),"\n" }
```

`$x` is private to the subroutine `foo` but it maintains its value between calls to the subroutine. A more well-behaved way to write the above would be:

```
{
    my $x;
    sub foo {
        return ++$x;
    }
}
```

or even to use a closure. But C does not allow us to declare bare blocks in the same way as Perl does; the C equivalent would look like this:

Example 1-7. Static variables

```
#include <stdio.h>

int foo () {
    static int x = 0;
    return ++x;
}

int main(int argc, char** argv) {
    int i;

    for (i=1; i<=10; i++)
        printf("%d\n",foo());

    return 0;
}
```

There are a few things to notice here:

- To maintain a variable's state between calls, declare the variable **`static`**.
- There is no range operator, nor a special variable like `$_`, nor is there a one-argument `foreach` loop to loop over an array; we only have the three-argument `for`.
- Calls to functions we declare *must* contain brackets, even when they take no parameters; otherwise, we end up taking the address of the function! (It's the difference between **`&mysub`** and **`\&mysub`**.)
- If the code inside a `for` loop is only one statement, we do not need to enclose it in a block. If it's any more than one statement, it must be enclosed in curly braces. This also applies for the other control structures, `if` and `while`.

C has no equivalent to Perl's dynamic scoping; (variables with `local`) this is something that the Perl internals attempt to get around, and we'll see how that's done in later chapters.

1.6. Data Types

If `argc` is an integer and `argv` is an array of strings, what other data types do we have? Perl only supports three types of variables (ignoring globs for now): scalars for single data items, arrays for storing many scalars and hashes for keyword/scalar pairs. A scalar is simply a *thing* that can be passed around and processed but the type of the scalar is not important. In C, the compiler needs to know whether you are using a number or a character and the type of number you are using (integer, floating point number, double precision floating point number). C supports quite a variety of data types which can vary from machine to machine, and the Perl internals define some special types which give us a machine-independent environment.

1.6.1. C Types

First, let's look at the basic C types:

1.6.1.1. int

The int type represents positive or negative integer values. C's data types are defined primarily, however, not by what they hold but by how many bits are used to represent them, since this defines their limits.

For example, an int on my machine is represented using 32 bits of memory. One of these is used as a "sign bit" to determine whether or not the value is positive or negative, and the other 31 bits are used to store the number; this means it has a range from -2147483647 to 2147483647. We can tell C to not use a "sign bit" and have all 32 bits available for storage by declaring an unsigned int, giving us a range from 0 to 4294967295.

I say “on my machine” because the size of these types is not guaranteed, nor is it defined by the C language; a compiler for a 64-bit processor may choose to use 64 bits to represent an int, although it may not. (This is why the Perl internals define their own types, to guarantee sizes.) There are a number of ways to determine the limits. The easiest is to set all the bits in an unsigned variable to 1, and examine the number produced. Just as in Perl we can set all the bits to 1 using the bitwise NOT operator, **~**:

```
#include <stdio.h>

int main (int argc, char** argv) {
    unsigned int i = ~0;

    printf("i is %u\n", i);
    return 0;
}
```

(Note that we use **%u** as the `printf` format specifier for an unsigned integer - this is the same as in Perl.)

This tells me that the highest unsigned integer is 4294967295, and so the highest signed integer must be one less than half of 1+4294967295.

The other method is slightly more complex, but more flexible. The header file `limits.h` defines some constants (See Section 1.9> for how this happens) which tell us the limits of the various sizes:

Example 1-8. Finding limits with `limits.h`

```
#include <stdio.h>
#include <limits.h>

int main (int argc, char **argv) {
    printf("The maximum signed integer is %d\n", INT_MAX);
    printf("The minimum signed integer is %d\n", INT_MIN);
    printf("The maximum unsigned integer is %u\n", UINT_MAX);
    /* UINT_MIN is not declared because it’s obviously 0! */
    return(0);
}
```

Produces:

```
The maximum signed integer is 2147483647
The minimum signed integer is -2147483648
The maximum unsigned integer is 4294967295
```

You should note that the `POSIX` module in Perl can also define these constants: (And notice how similar Perl with **`use POSIX`** can be to C!)

```
use POSIX;
```

```
printf("The maximum signed integer is %d\n", INT_MAX);
printf("The minimum signed integer is %d\n", INT_MIN);
printf("The maximum unsigned integer is %u\n", UINT_MAX);
```

1.6.1.2. char

Characters are nothing more than numbers. A character, to C, is merely an eight-bit integer; depending on your architecture and compiler, it may be signed and thus would have the range -128 to 127. Since we're more used to thinking of characters running from character 0 to character 255, we can use unsigned char to get that range.

Since characters are just numbers, a single-quoted character in C acts like the `ord` operator in Perl - it produces an integer representing the character set codepoint of that character:

```
#include <stdio.h>

int main(int argc, char** argv) {
    printf("%d\n", '*');
    return 0;
}
```

This is equivalent to Perl's **`print ord("*"),"\n"`**. Similarly, we can turn numbers into characters with the `printf` format specifier **`%c`**, just as in Perl:

Example 1-9. Manipulating char data

```
#include <stdio.h>

int main(int argc, char** argv) {
    unsigned char i;

    /* print "Character $_ is ", chr($_), "\n" for 0..255; */
    for (i=0; i < 255; i++)
        printf("Character %d is %c\n", i, i);

    return 0;
}
```

Why do we say **`i < 255`** instead of **`i <= 255`**? Because of the way `i` is stored, it can only ever be between 0 and 255, so our termination clause is useless. When a C value overflows the storage of its variable, it "wraps around" - the higher bits are truncated. For instance:

Figure 1-1. Overflowing storage

```
                    |87654321| - 8 bits of an unsigned char
                    | Binary |
i = 254             |11111110|
```

```
i++                |11111111|
i = 255            |11111111|
i++               1|00000000| - Overflow
i = 0              |00000000| - Truncation
```

So, for an unsigned char, 255+1 = 0, and since 0 is less than or equal to 255, our program would never have terminated.

1.6.1.3. short

Sometimes an int stores more bits than you need, so you may want to use a smaller type. A short (or short int) is usually half the size of an int: the limits `SHRT_MAX` and `SHRT_MIN` should tell you the size.

```
printf("The maximum short is %d\n", SHRT_MAX);
printf("The minimum short is %d\n", SHRT_MIN);
```

(And you can run that in C or Perl at your choice...)

shorts are available in signed and unsigned flavours, but are only rarely used.

1.6.1.4. long

For representing larger numbers, longs (long ints) are sometimes available. On some machines, these are twice the width of an int; however, on many machines longs and ints are equivalent. The new C standard, C99, also allows long long ints, which are twice as wide again. Both types are available as unsigned variants.

1.6.1.5. float

The world is not purely made up of integers; there are also floating point values, and these need a separate data type to hold them. (Perl is happy to have us put strings, integers and floating point values in the same scalar, but C forces us to split them up.)

Floating point types are always signed in C, and a floating point value is represented by two numbers, the *exponent* (`e`) and the *mantissa* (`m`) such that the value to be stored is >. The choice of the number of bits for exponent and mantissa determines the accuracy and the range of the type. It is important to remember that a floating point number can not represent every number with complete accuracy. Some numbers (for example `1/3` and `1/7`) can never be represented perfectly regardless of the number of bits in the float and this has to be carefully considered if accuracy of numerical calculations is important to you.

You must also carefully consider the difference between floating point operations and integer operations. Consider the following program:

```
#include <stdio.h>

int main (int argc, char** argv) {
  float fraction1 = 1 / 4;
  float fraction2 = 1 / 4.0;
  printf("%f %f\n", fraction1, fraction2);

  return 0;
}
```

When you run this program you may be surprised to get the following output:

```
0.000000 0.250000
```

Here `fraction2` correctly has a value of `0.25` but `fraction1` has a value of `0.00`. This seeming inconsistency is a product of the compiler; when the compiler sees literal numbers it has to assign a type to them. As there is no indication to the contrary, numbers that look like integers are assigned to integer types and numbers that look like floating point numbers are assigned floating point types. This means that the compiler translated the above assignments to

```
float fraction1 = (int)1 / (int)4;
float fraction2 = (int)1 / (float)4.0;
```

When there is an arithmetic operation between two variables of different types the compiler converts the variables to the highest type using the rules given in Section 1.7>. In this case, and integer is converted to a float when combined with another float:

```
float fraction1 = (int)1 / (int) 4;
float fraction2 = (float)1.0 / (float)4.0;
```

Now for the trick: the division operator performs integer division (that is, effectively, `int(1/4)`) when both of its operands are integers, and floating point division when its operands are floats. Hence, the value that's stored into `fraction1` is the result of *integer* division of `1` and `4`, and the value that's stored into `fraction2` is the floating point division which keeps the fractional part.

The moral of the story is: if you want a floating point result from your code, make sure that at least one of your operands is a float.

1.6.1.6. double

doubles are, very simply, high-precision floats; they contain a larger exponent, although how much larger is unspecified - on this system, doubles can range from `2.2250738585072e-308` all the way up to `1.79769313486232e+308`; that's 10 bits of exponent and 53 bits of mantissa.

On some systems, long doubles may be available for even more bits.

1.6.1.7. void

void is a special data type which is used, as we've seen in Example 1-6> above, to indicate that a function has no return value and should be called in void context.

As we'll see in the next chapter, the void type is also used as a "generic type" to enable data of any type to be passed to and from a function.

1.6.2. Types defined in Perl

To get around the implementation-specific nature of the limits of the basic C types, Perl defines a number of types which are guaranteed to have certain properties. Perl also defines a number of far more complex types, which we'll look at in Chapter 3, which allow us to represent Perl scalars, hashes and so on. For now, we'll examine the simple types from which almost all variables inside Perl are made up. For guaranteed portability, you should use these types when your code interfaces with Perl, rather than the types above.

1.6.2.1. I8, I16, I32

This set of types are used for different sizes of integers, and have the property that they are guaranteed to hold *at least* the number of bits that their name implies: an I8 will definitely hold 8 bits, (and might hold more) and so can be used to store values from -128 to 127. An I8 is almost always equivalent to a char.

Each of these types has a corresponding unsigned type: U8, U16, and U32. On 64-bit machines, I64 and U64 are also available.

You may ask how these types are defined: C has a special convention for defining types, the **`typedef`** operator, which allows us to provide aliases for type names. Here is Example 1-9> reimplemented using **`typedef`** and U8.

Example 1-10. Using `typedef`

```
#include <stdio.h>

typedef U8 unsigned char;

int main(int argc, char** argv) {
    U8 i;

    for (i=0; i < 255; i++)
```

```
            printf("Character %d is %c\n", i, i);

        return 0;
    }
```

1.6.2.2. IV, UV

An IV, and its unsigned counterpart UV, is the type used to represent integers used in a Perl program. When you say **`$a = 123456;`**, the `123456` is stored in an IV or UV. The reason why Perl uses an IV rather than any guaranteed-size types of the previous section is that IV provides another guarantee: it's big enough to be used to store a *pointer*, which is the C equivalent of a reference. (We'll look into pointers in a lot more detail in the next chapter, and see how they relate to Perl references in Chapter 3.)

1.6.2.3. NV

An NV is the type used to represent floating-point numbers in a Perl program; once again, this is guaranteed to be able to store a pointer, although it's hardly ever used to do so.

1.6.2.4. STRLEN

Finally in our tour of types, STRLEN is an unsigned integer type which tells us how big something is in bytes; it's usually used to represent the size of a string.

1.7. Casting

C uses very simple rules to convert values between types: data will be converted when it moves from a 'smaller' type to a 'bigger' type, such as from an int to a float, but when converting back down, only the portion of the representation which 'fits' in the smaller type will be retained. (This may or may not trigger a warning from the compiler about information being lost.) For instance, the following code:

```
int x = INT_MAX;
short int y;
y = x;
```

will leave `y` equal to `-1`, because all of the bits in the smaller type will be set.

The conversion happens implicitly when two differently-typed values are the operands to the arithmetic operators, or when one assigns one value to a variable of different type; it also happens when the type of a value passed as a function parameter is different to what the function's prototype says it should be.

You can also force the explicit conversion to one particular type by the use of the *cast* operator; simply put the target type in parentheses before the expression you wish to cast. This is rarely used for real values, but is extremely important when we come to examine pointers and structures.

1.8. Control Constructs

Because Perl borrowed its syntax heavily from C, C's control constructs should be familar to you: we have **`if (...) {...}`**, **`while (...) {...}`**, **`do { ... } while (...)`** and **`for(;;;)`**, and these all work the same way they do in Perl.

1.8.1. Statements and blocks

One difference, however, is that one may omit the braces from a block under a control construct if that block contains only one statement. For instance, we may write:

```
if (a > b) {
    max = a;
} else {
    max = b;
}
```

as:

```
if (a > b)
    max = a;
else
    max = b;
```

A control construct itself counts as "only one statement", so we may also write such things as this program, to count the number of printable characters in the character set:

```
#include <stdio.h>
#include <ctype.h>

int main(int argc, char **argv)
{
    unsigned char i;
    int printables = 0;

    for (i=0; i<255; i++)
        if (isprint(i))
            printables++;

    printf("%i printable characters\n", printables);
    return 0;
```

```
}
```

The function `isprint`, whose prototype is in `ctype.h` tells us whether or not a character, in the range 0 to 255, is printable.

When using nexted control structures without braces like this, it's important to be aware of the '*dangling else*' problem; that is, in the following code, which **`if`** does the **`else`** belong to?

```
if (utf)
   if (!haslen)
       len = getlength(string);
else
   len = 1;
```

Well, the indentation shows us what we *mean* - we want `len` to be `1` if `utf` is not set - but that's not how the compiler sees it; what really happens looks like this:

```
if (utf)
   if (!haslen)
       len = getlength(string);
   else
       len = 1;
```

Editors such as Emacs will automatically indent code to the correct column when you press **Tab**, but it's best to use braces in such cases anyway to reduce confusion.

1.8.2. `break` and `continue`

Perl's **`last`** control statement is spelt **`break`** in C. Here's a function which cuts off a string after the first space character:

Example 1-11. `break`ing out of a look

```
int token (char s[]) {
    unsigned int len = strlen(s);
    int i;

    for (i=0; i < len; i++)
        if (isspace(s[i]))
            break;

    s[i] = '\0';
    return i;
}
```

When a whitespace character (space, tab, new line, etc.) is found, the **`break`** statement makes C immediately leave the **`for`** loop, and makes that character the end of the string. We return the character offset as the new length of the string; in the next chapter, we'll see how we can use this to get at the rest of the string.

Similarly, **`next`** is replaced by **`continue`**; this fragment of code (modified from `sv.c` in the Perl core) only processes non-zero elements in an array:

```
for (i=0; i > oldsize; i++) {
     if (!ary[i])
         continue;
     curentp = ary + oldsize;
     ...
}
```

There is no equivalent to Perl's **`redo`**. If you really need it, you can use **`goto`** and labels just like in Perl, but, just like in Perl, ten times out of ten, you don't.

1.8.3. `switch`

One control structure that Perl doesn't have[2] is the **`switch`** statement. This allows you to test an expression against multiple (constant) values. It's a lot easier than using **`else if`** over and over again. Here's an example from the Perl core, when Perl has seen one of the **`-X`** file test functions, and is trying to work out which one you mean. It has the next character in `tmp`, and is choosing from a number of constants to set the value of `fstest`:

```
  switch (tmp) {
     case 'r':
               ftst = OP_FTEREAD;
               break;

     case 'w':
               ftst = OP_FTEWRITE;
               break;

     case 'x':
               ftst = OP_FTEEXEC;
               break;

     case 'o':
               ftst = OP_FTEOWNED;
               break;
     ...
  }
```

Notice that we **`break`** after every single case; this is because **`switch`** is, in fact, a glorified computed **`goto`**. If we don't break, the program control will "fall through" to the next case:

```
int i = 0;

switch (i) {
   case 0:
            printf("It's 0\n");
   case 1:
            printf("It's 1\n");
   case 2:
            printf("It's 2\n");
}
```

The above code will actually execute *all three* print statements. Sometimes this really is what we want, but we should take care to mark the "fall through" if we're likely to forget it. Here's an example of how fall through could be useful. Note that we're actually falling through the cases which have no statements as well; if the character is `1`, then we fall through cases `2`, `3`, ... `7`.

```
 switch (c) { /* "c" is some character"
    case '0': case '1': case '2': case '3':
    case '4': case '5': case '6': case '7':
              could_be_octal = 1;
              /* Fall through */

    case '8': case '9':
              could_be_dec = 1;
              /* Fall through */

    case 'A':
    case 'B':
              /* This is actually fall through, too */
    case 'C':
    case 'D':
    case 'E':
    case 'F':
              could_be_hex = 1;
  }
```

1.9. Macros and the C Preprocessor

Before your C code reaches the compiler, it goes through an intermediary program, the *preprocessor*. We've already said that header files such as `stdio.h` and `ctype.h` contain function prototypes - the preprocessor is responsible for inserting the content of those files into the current program. This is done with the **`#include`** preprocessor directive:

```
#include "header.h"
```

will insert the contents of the file `header.h` from the current directory into the copy of your source code that is passed to the compiler.

Why did we have this:

```
#include "header.h"
```

but previously this?

```
#include <stdio.h>
```

The difference is that by using quotes, we are telling the preprocessor that the file `header.h` is in our current directory, but that `stdio.h` is somewhere in the system and it should go and look for it; the preprocessor has a built-in search path which includes the locations of the headers for the standard library.

Header files may themselves use **`#include`** directives to pull in other files; if we look at the pre-processed output to our “hello world” example, Example 1-1>, we would see that many different header files have been included:

```
# 1 "/usr/include/stdio.h" 1 3
# 1 "/usr/include/features.h" 1 3
# 142 "/usr/include/features.h" 3
# 208 "/usr/include/features.h" 3
# 1 "/usr/include/sys/cdefs.h" 1 3
# 65 "/usr/include/sys/cdefs.h" 3
# 283 "/usr/include/features.h" 2 3
# 1 "/usr/include/gnu/stubs.h" 1 3
# 311 "/usr/include/features.h" 2 3
# 27 "/usr/include/stdio.h" 2 3
...
```

As well as including header files, we may use the preprocessor to define *macros*, pieces of text which will be substituted in our source. The syntax for a macro is:

```
#define text replacement
```

For instance, we may use macros to give meaningful names to particular constants:

```
#define MAX_RECURSE  64
#define FAILURE      -1

int recursive ( ... ) {
    static level=0;
```

```
    if (level++ > MAX_RECURSE) {
        printf("! Maximum recursion level reached!\n");
        level--;
        return FAILURE;
    }

    /* Do something here */

    return level--;
}
```

Macros may also be given arguments, so they appear to be like functions; for instance, the `isspace` we used in our **`break`** example, Example 1-11>, is typically defined as a macro, like this:

```
#define isspace(c)  ((c) == ' ' || (c) == '\t' || (c) == '\n' || (c) =='\r' || (c) == '\f')
```

The replacement text will be placed into our program wherever we use `isspace`, and the `c` is replaced by whatever argument we gave to it. Thus, when our example reached the compiler, it probably looked a little more like this;

```
int token (char s[]) {
    unsigned int len = strlen(s);
    int i;

    for (i=0; i < len; i++)
        if (((s[i]) == ' ' || (s[i]) == '\t' || (s[i]) == '\n' || (s[i]) =='\r' || (s[
            break;

    s[i] = '\0';
    return i;
}
```

Because the text is literally replaced, we must be very careful when calling macros (or things we suspect to be macros) when using side-effects such as post-increment. If we'd said, for instance,

```
while (i < len)
    if (isspace(s[i++]))
        break;
```

the compiler would see

```
while (i < len)
    if (((s[i++]) == ' '  ||
         (s[i++]) == '\t' ||
         (s[i++]) == '\n' ||
         (s[i++]) == '\r' ||
         (s[i++]) == '\f'))
        break;
```

and `i` would be incremented rather a lot faster than we want.

As that line is very long, we may use backslash (\) characters to break it up over several lines; when the preprocessor sees a backslash at the end of a line, it is removed and the next line is concatenated. Hence, this is equivalent:

```
#define isspace(c)  ((c) == ' '  || \
                     (c) == '\t' || \
                     (c) == '\n' || \
                     (c) == '\r' || \
                     (c) == '\f')
```

Perl makes extremely heavy use of macros internally.

1.10. Library Functions

As we've already mentioned, C provides a standard library of useful functions - things like `printf` and `isspace`. However, the effects and side-effects of these library functions can vary between systems, so Perl reimplements or redefines the useful portion of the standard library internally.

The file `pod/perlclib.pod` in distributions after 5.6.0 and 5.7.0 contains a table of equivalents between the sort of functions that you'd expect to see in an ordinary C program, and the functions that should be used in XS, embedding and the Perl internals, which you'll see in later chapters.

1.11. Summary

We've seen an introduction to the C language from a Perl programmer's point of view. The most important things you should take from this chapter are:

- In C, everything is a function, even the **`main`** body of your program, and all functions must have prototypes before they are called.

  C has four different types of variable: parameter, automatic, global and static. All variables *must* be declared before being used.

  C is a *strongly typed* language; it has a number of data types, and each variable can have one and only one type.

  C has no built-in functions whatsoever, but it does have a standard library of functions available; however, as we'll see in later chapters, when using Perl and C, you're likely to use Perl's reimplementations of the standard functions.

  C's syntax is very similar to Perl's, but is more 'austere': no statement modifiers, no **`foreach`**, no **`redo`**, but it does have a **`switch`**.

  C programs are automatically run through a preprocessor which can be used to give meanings to tokens and create in-line functions.

Notes

1. For historical reasons - `a.out` was the name of a particular type of executable file format.
2. Although Damian Conway's `Switch` module provides an implementation

Chapter 2. Introduction to XS

This chapter will introduce the fundamentals of interfacing perl to other programming languages. Before we can describe how to do this we first have to 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 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 (http://www.cpan.org)* (CPAN).

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 this is not done, it would be possible for your module to clash with other modules or with the main program and this is undesirable. There are three main ways that this is achieved:

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

Here is an example of a minimalist module that shows how these constraints can be implemented:

```
package Example;                                    ❶

use 5.006;                                          ❷
use strict;                                         ❸

use base qw/Exporter/;                              ❹

our $VERSION = ’1.00’;                              ❺
```

```
our @EXPORT_OK = qw/ myfunc /;                    ❻

# Code
sub myfunc { 1; }                                 ❼

1;                                                ❽
```

❶ This is the namespace declaration. All code in this file is only visible in this namespace unless explicitly referred to from outside or until a new `package` function is found.

❷ This line makes sure that the perl version used by this module is at least version 5.6.0 (5.006 in the old numbering style). This is because the module makes use of the `our` variable declaration that was introduced in this version of Perl

❸ All Perl modules should have `strict` checking. Among other things, this pragma instructs Perl to tell you of any undeclared variables that it comes across; an excellent way to minimize obvious bugs in code.

❹ Here we inherit methods from the `Exporter` class in order to enable exporting of functions and variables to the namespace that `use`es this module.

❺ This line defines the version number of the module. It is used by CPAN for indexing and to enable Perl to check that the correct version of a module is available.

❻ The `@EXPORT_OK` array contains a list of all the functions 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 is not desirable in most cases.

❼ This is the code that implements the module functionality. The actual code for the module goes here.

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

If you 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 you load it as

```
use Example;
```

the function `myfunc` will not be imported but can still be accessed as `Example::myfunc()`. More information on perl modules can be found 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 by 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 the CPAN) Perl provides a framework that can be used to automate installation. In order to use this framework we need to create a number of files in addition to the module:

`README`

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

`Makefile.PL`

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 when run [1]. It is this makefile that is used to build, test and install the module. A Perl module installation usually consists of:

```
% perl Makefile.PL                                    ❶
% make                                          ❷
% make test                                ❸
% make install                                  ❹
```

❶ Generate the make file

❷ use the `Makefile` to build the module.

❸ Run any included tests.

❹ Install the module into the standard location.

The `Makefile.PL` is useful since it deals with all the platform specific options that are required to build modules. This system guarantees that modules are built using the same parameters that were used to build Perl itself. This platform configuration information is stored by Perl in the `Config` module.

At its simplest the `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 of the system dependent functionality is provided by the `ExtUtils::MakeMaker` module. The `WriteMakefile` routine accepts a hash that controls the contents of the makefile. In the above 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 itself and PREREQ_PM lists the dependencies for this module (the `CPAN` module uses this to determine which other modules should be installed). Additional options will be described in later sections and a full description can be found in the documentation for `ExtUtils::MakeMaker`.

`MANIFEST`

A list of all the 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 for creating a distribution file when using "**`make dist`**".

`test.pl`
`t` directory

Although not a requirement, all module distributions should include a test suite. Test files are very important for testing the integrity of the module. They can be used to make sure that the module works now, that it works after improvements are made and that it works on platforms that may not be accessible by the module author. When Perl sees a `test.pl` file in the current directory the resultant `makefile` includes a test target that will run this file and check the results. A good test suite is one that uses all the functionality of the module in strange ways. A bad test suite is one that simply loads the module and exits (or, even worse, no test suite at all). 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`, `ftp.t`). Test programs are written using the framework provided by the `Test` module. A simple example, based on `Example` could be:

Example 2-1. Simple test program

```
use strict;
use Test;                                                    ❶

BEGIN { plan tests => 2 }                                    ❷

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

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

❶ Load the testing framework.

❷ Inform the test framework to expect two test results.

❸ Load the module that is being tested and import the required routines.

❹ The `ok` takes the supplied argument and checks to see whether the argument is true or false. In this case the argument is always true and the `ok` always prints an `ok` message so long as the module has been loaded successfully. If the module fails to load this line will never be reached.

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

Templates for these files can be created by using the **h2xs** program that comes as part of perl. When used with the `-X` option a basic set of files are created:

```
% 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 above **h2xs** generates a file called `Changes` that can be used to track changes that are made to the module during its lifetime. This information is useful for checking what has happened to the module and some editors (e.g. emacs) provide an easy means of adding to these files as the code evolves.

2.2. Why interface to other languages?

The flexibility of the Perl language means that it is possible to solve many programming problems without ever needing to use any other language. There are many modules on CPAN that are extremely useful and are written completely in Perl.

In fact, writing a module in Perl has a number of advantages over using other languages:

- It is far easier to write portable cross-platform code in perl than it is to use C. One of the successes of perl has been its support of many varied operating systems. It is unlikely that a module written in C could be as portable as a perl version without expending much more effort.
- Some problems do not need the speed gain or added complexity that will come from using a compiled language. There are many programmers that are proficient in Perl (and/or Java) for whom writing a perl module would be much more efficient (in terms of programming effort) than writing the equivalent in C.

People program in perl for a reason and this should not be forgotten when it comes to deciding on reasons for not using perl. These issues were addressed in the development of the standard `File::Temp` module (part of Perl 5.6.1). This module provides a standard interface for creating temporary files from Perl. The original intention was that this module would be written in C but it quickly became apparent that a perl implementation would be easier because of portability problems (since it was to be a standard module it would have to work on all platforms supported by Perl) and speed would not be a real issue unless thousands of temporary files were required.

Having addressed why not to use another language there are, of course, two important reasons why another language is required:

Speed

In some cases Perl is simply too slow for a particular program. In this case the choice is either to change the algorithm or to use a faster programming language. The Perl Data Language was created specifically to address the case of processing N-dimensional arrays but there are still times when another language is required.

Functionality

There are many useful libraries available that have been written in other languages (especially C, and for numerical applications, Fortran). If new functionality is required that is present in an external library then it is usually far better to provide a perl interface to the library than to recode the library in perl.

2.3. Interfacing to another language - C from XS

Now that we have covered how to create a module and why we need to interface to other languages, this section will describe the basics of how to combine C code with Perl. We start with C because that 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. Interfacing to C using other techniques (e.g. SWIG and the `Inline` module) will be described 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 the function and the variables that are to be returned. The XS file is translated to C code that can be understood by the rest of the perl internals by the XS compiler (called **xsubpp**). 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 on how to turn a perl scalar variable into a C integer.

This section will begin by describing the changes that have to be made to a standard Perl module in order to use XS and will then proceed with an example of how to provide simple C routines to Perl.

2.3.1. 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 is used to indicate to **h2xs** that constant autoloading is not required; more on that later (see Section 2.4.3). The `-n` is used to specify the name of the module in the absence of a C header file. Besides the creation of the `Example.xs` file the only change made to the files generated previously is to change the perl module (the `.pm` file) so that it will load the C code that is to be generated. The module created by **h2xs** has many features that are not important for this discussion so we will start from the minimalist module described in Section 2.1 and modify it to support shared libraries:

```
package Example;                                    ❶

use 5.006;
use strict;

use base qw/Exporter DynaLoader/;                   ❷

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

bootstrap Example $VERSION;                         ❹
1;
```

❶ Start a new package (namespace) named `Example`.

❷ This class now inherits from both `Exporter` and `DynaLoader`.

❸ The `print_hello` function is only referenced in the export list; the shared library is responsible for supplying all the information required by perl to implement this function.

❹ Load the `Example` shared library associated with the provided version number.

There are only two changes: The first is that the module now inherits from the `DynaLoader` module as well as the `Exporter` module. The `DynaLoader` 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 [2]. The second change is the line added just before the end of the module. The bootstrap function is the one that does all the work of loading the dynamic library named `Example` making sure that the 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);`**

2.3.2. The XS File

Now that the preliminaries are taken care of the XS file itself 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 start 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 always have to 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 start by trying to extend Perl to include functions described in Chapter 1. These must either be added to the `.xs` file directly or included from either a specially built library or a separate file in the distribution. For simplicity we will start by adding the code from Example 1-1 and Example 1-3 directly to the `.xs` file:

```
#include <stdio.h>

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

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

❶ Here we have replaced the `main` declaration of Example 1-1 with a new function name that can be called from elsewhere.

The XS part of the file is indicated by using the `MODULE` keyword. This 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 functions is also defined on this line. This allows 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 themselves can be added.

2.3.3. Example: "Hello, world"

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

```
void                                                    ❶
print_hello()                                           ❷
```

❶ Type of value returned to perl from this function. In this case nothing is returned so the value is void

❷ Name of the function as seen from perl and the arguments to be passed in.

An XS function 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. This function is very simple so XS needs no extra information to work out how to interface perl to the `print_hello` function.

Your XS file should now contain the following:

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

#include <stdio.h>

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

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

MODULE = Example  PACKAGE = Example

void
print_hello()
```

If this file is saved as `Example.xs` the module can be built in the *normal* way (don't forget to add this file to the `MANIFEST` if it is not there already):

Example 2-2. Output from build of 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 -march=i386 -mcpu=i686 -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 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 The first step is to create the directory structure that will receive the module files as the build proceeds. This directory is called `blib` for "Build Library".

❸ This copies all the perl files to the architecture independent directory.

❹ This line runs the XS compiler and translates the XS file to C code. The compiled file is written to a temporary file and then moved to `Example.c` rather than writing straight to the C file. This is done to prevent partially translated files being mistaken for valid C code.

❺ This warning can be ignored. It is informing you that some XS functions were defined without specifying a perl prototype. This warning can be removed 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 then 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 the `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 method for this is platform dependent and the methods are retrieved from the `Config` module.

* *We also need to grab the output of the same thing on Windows NT to show how similar it is and where the differences lie*

When the **Makefile.PL** is run it now finds a `.xs` file in the directory and modifies the resulting makefile to process that in addition to the perl module. We don't yet have a test program but we can test it from the command line to see what happens:

```
% 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 in 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 since 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. This is required because the `Example` module has not yet been installed. Running tests like this is not very efficient or scalable so the next step in the creation of this module is to write a test program. The testing framework provided by the `Test` module [3] makes this easy. Here the test program from Example 2-1 has been modified to test our `Example` module:

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

ok(1);
print_hello;
ok(1);
```

If the above program is saved to a file named `test.pl` we can then use the make program to run the test[4]

```
% 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
ok 1
```

```
hello, world
ok 2
```

The problem with the simple test above is that it is not really testing the `print_hello` subroutine but simply whether (a) the module has loaded (b) the `print_hello` subroutine runs without crashing. Whilst these are useful tests to do they do not tell us anything about the subroutine itself. This is 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 okay. The next section will fix this by adding return values.

2.3.4. Return Values

Adding a simple return value to an XS routine (i.e. 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 this by adding this function to the top of our XS file:

Example 2-3. "Hello, world" with a return value.

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

Here we have added a new function with a slightly different name to indicate that we are now returning an integer value. This makes use of the fact that the `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()
```

Here the function is absolutely identical to the XS code for `print_hello` except that the void declaration has now been changed to an int. Once this is saved it can be rebuilt simply by typing **`make`** as before. If we now modify the test script to add

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

and change the planned number of test to three we can now test that the routine is returning 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'
```

2.3.5. Arguments and return values

Our `treble` function from Chapter 1 takes an integer argument and returns an integer. This would be represented in XS as follows:

Example 2-4. XS for the treble function

```
int                                                    ❶
treble( x )                                            ❷
  int x                                                ❸
```

❶ Returns an integer

❷ This is the signature of the command as it will be visible to Perl. There is now an input argument.

❸ All the arguments listed in the previous line are then typed in this and successive lines. For simple C types Perl knows the translation without having to be told.

The example from Chapter 1 could now be written as:

```
% cat treble.pl
use Example;
```

```
print "Three times ten is ", Example::treble(10), "\n";
% perl -Mblib treble.pl
Three times ten is 30
```

2.4. Taking things further

So far we have shown how to use XS to provide wrappers to simple C functions with simple arguments where we want the signature of the Perl subroutine to match the signature of the C function. In many cases this is too simplistic an approach and extra code must be supplied in the XS layer. The XS wrapper allows C code to be provided 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 Example 2-3 and Example 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
```

Here 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 the return value is not automatically configured as a return value. **xsubpp** needs to be told explicitly that `RETVAL` should be returned and this is done by using the `OUTPUT` keyword.

2.4.1. Modifying input variables

In some cases the 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 output. The `OUTPUT` keyword is used for this. 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
```

which is equivalent to this Perl subroutine:

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

or more pedantically:

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

and suffers from the same problem - the input argument must be a variable and not a constant else it can not be modified. If a constant is passed in (for example a straight number as in out 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.

2.4.2. Output arguments

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

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
```

Here, 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 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 that onto Perl. In some cases this is desirable, maybe for familiarity with existing library interfaces, but in other cases not so. This brings us to the question of interface design and that topic is addressed in Section 2.6

2.4.3. 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` options. The approach taken by **h2xs** is to use the `AutoLoader` module to determine the value of constants on demand at runtime rather than importing every constant when the program starts.

> An extreme example is the standard `POSIX` module. This defines over 350 compiler constants and creating this many subroutines during loading would impose a large overhead.

The autoloading is implemented in two parts. Firstly an `AUTOLOAD` 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;                                              ❶
use AutoLoader;                                         ❷
use Carp;                                               ❸

sub AUTOLOAD {                                          ❹
    my $sub = $AUTOLOAD;                                ❺
    (my $constname = $sub) =~ s/.*::///;                ❻
    my $val = constant($constname);                     ❼
    if ($! != 0) {                                      ❽
        if ($! =~ /Invalid/ || $!{EINVAL}) {            ❾
            $AutoLoader::AUTOLOAD = $sub;               ❾
            goto &AutoLoader::AUTOLOAD;                 ❾
        } else {
            croak "Your vendor has not defined constant $constname";(10)
        }
    }
```

```
    {
        no strict 'refs';                                   (11)
        *$sub = sub () { $val };                            (11)
    }
    goto &$sub;                                             (12)
}
```

❶ We use this module so that we can check for explicit values of the `errno` variable.

❷ Load the `AutoLoader` module. This is only required if you want to dynamically load additional functions from disk.

❸ Load the `Carp` module. This imports the `croak` function.

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

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

❻ This runs 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 code generated by **h2xs** passes a second argument to this routine (`$_[0]`) but for simple constants it can usually be removed from the routine.

❽ This checks the error status from the `constant`. 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 behaviour 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` sets `errno` if the requested constant can not be located.

❾ These lines check to see if `errno` has returned with errors that indicate that the constant does not exist at all. In that case it passes control to `AutoLoader` in case some functions are to be autoloaded from `.al` files. These checks are only required if auto-loaded routines are expected, else this is simply an extra overhead for the program.

(10) If the constant should be available but was not defined this line stops the program. `croak` is used rather than `die` so that the line number in the callers code is printed in the error message rather than the line number in the `AUTOLOAD` subroutine.

(11) At this point in the routine the value of the constant has been determined and, in principle, that value could be retuned to the caller. Whilst it is valid to do that 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 single time the constant is requested; very inefficient. To overcome this inefficiency the `AUTUOLOAD` 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 4.5 for more explicit details of how this can work). The name of the glob is stored in `$sub` and therefore requires that soft references are allowed. The **`no strict 'refs'`** turns off strict checking to allow this. If you are uncomfortable with glob assignments the same effect can be achieved simply by using a string `eval`: **`eval "sub $sub () { $val }";`**.

(12) 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` was 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;                                                 ❷
  switch (*name) {                                           ❸
  case 'S':                                                  ❹
     if (strEQ(name, "S_ISGID"))                             ❺
#ifdef S_ISGID                                               ❻
         return S_ISGID;                                     ❻
#else
  goto not_there;                                            ❻
#endif
  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;                                                   ❼
    }
    errno = EINVAL;                                          ❽
    return 0;

not_there:                                                   ❾
    errno = ENOENT;
    return 0;
}

MODULE = Fcntl_demo PACKAGE = Fcntl_demo                     (10)

IV                                                           (11)
constant(name)
   char * name
```

❶ 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)

❷ Sets `errno` to zero (no error) so that its value can be checked on exit from the function.

❸ This denotes the start of a block that will switch on the first character of the requested constant name.

❹ This block is executed if the constant name starts with a "S".

❺ Compare the requested name with the string "`S_ISGID`"

❻ This is the block that does all the work. 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.

❼ If the constant can not be found even though it started with the correct letter the switch is exited since there is no chance of it matching the remaining tests.

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

❾ 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.

(10) This defines the start of the XS part of the file.

(11) The XSUB definition for `constant` is very simple with a single string argument and a return type of IV.

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

2.4.3.1. ExtUtils::Constant

From perl 5.8.0 a new module has been added, `ExtUtils::Constant`, that 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 has a number of advantages over the current scheme:

- Improvements in the constant handling code can be made without having to touch every single module that uses constants.
- The XS files are now much simpler. No longer are the files dominated by long repetitive lists of constants and C pre-processor directives.
- The new system allows compiler constants to have different types. An integer constant is treated differently to a floating point constant.

2.5. What about that `Makefile.PL`?

So far we have not really addressed the contents of the file that is instrumental in configuring the build process. When building simple perl modules the `Makefile.PL` is almost empty providing just 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 building XS extensions since the makefile that is generated must include information on 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 has been handled automatically by the `WriteMakefile` function since it detects the presence of an XS file and sets up the appropriate makefile targets. However, this only works if the module can be built without additional configurations above and beyond that used to build perl originally. So far the examples have not required anything more than standard include files and libraries. What would happen if we were building a wrapper around a library that was not included by default? Add the following code to the XS example to find out. It should print out the version of the XPM library on your system. The include directive should go after the Perl includes and the XS declaration in the XS section.

```
#include <X11/xpm.h>

...

int
XpmLibraryVersion()
```

If the above XS code is added to our example file and built we get this:

```
% 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 out:

```
% perl -Mblib -MExample -e 'Example::print_hello'
Using ..../Example3/blib
hello, world

% perl -Mblib -MExample -e 'print Example::XpmLibraryVersion'
Using ..../Example3/blib
perl: error while loading shared libraries:
    /home/timj/perl/perlbook/book/Example3/blib/arch/auto/Example/Example.so:
    undefined symbol: XpmLibraryVersion
```

The above output indicates that the earlier routines (e.g. `print_hello`) still work but the new routine doesn't. The error message indicates that Perl could not find `XpmLibraryVersion` in any of the libraries that it has already loaded. This is not surprising since Perl is not linked against graphics libraries during a standard build. To overcome this problem we can use the `Makefile.PL` to provide the information necessary to locate the correct libraries. The `Makefile.PL` 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. The LIBS key can be used 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. [5] `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 have to change the LIBS entry so that the `Xpm` library will be included:

```
'LIBS' => '-L/usr/X11R6/lib -lX11 -lXpm', # all X11 programs require -lX11
```

Rebuilding the module now gives:

```
% 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 ❶
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 The value specifed for LIBS appears on this line

and the test runs as expected:

```
% perl -Mblib -MExample -e 'print Example::XpmLibraryVersion'
Using ..../Example3/blib
30411
```

Similarly, extra include paths (reference to chapter 1?) can be added using the `INC` key. 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 is usually of the form **`-I/some/dir`**.

2.5.1. It really is a Perl program

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

As an example let's suppose we wanted to build an interface to a Gnome library [6]. Most Gnome libraries come with configuration scripts that can be used to determine the required libraries and include directories and these must be used inside the `Makefile.PL` rather than hard-wiring the location of the Gnome system into the program.[7] To support this, the `Makefile.PL` 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 new lines
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' => $incs, # e.g., '-DHAVE_SOMETHING'
    'INC' => '', # e.g., '-I/usr/include/other'
);
```

2.6. Interface Design - Part 1

Now that we have seen how to create Perl interfaces to simple C functions and library routines this section will provide some advice on how these C routines should behave in a Perl world.

- When 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.
- In C arguments can be input arguments, return arguments or both and there is no way to distinguish this behaviour from the prototype. 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 take an input integer, store a value into a double (the asterisk indicates a pointer in C, we will talk about those in the next chapter) 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 only way to do this is to manipulate the argument stack by hand (described in Chapter 6). In newer versions of **xsubpp** it is possible to indicate that some arguments are to be returned differently using modifiers when declaring the function signature.

```
REQUIRE: 1.9508           ❶

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

❶ Make sure that 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 may want to consider removing it completely:

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

Returning `undef` if an error occurs. We'll see how to do this in Section 6.9.

- Do not ask the Perl programmer to provide information that Perl already knows. For example, the C function might need to know the size of a buffer being passed in. Since Perl already knows the length it is redundant and error-prone to ask the programmer to provide that information explicitly.
- When interfacing to a library do not blindly import every single function into Perl. It is possible that many of the functions are support functions needed by the C interface but irrelevant to Perl. Additionally many of the constants may not be needed.
- Make use of Perl namespaces. Many C libraries use the library name as a prefix to every function (for example, many functions in the Gnome (http://www.gnome.org) library start with `gnome_`) so use the package name to indicate that information and strip the common prefix. The `PREFIX` keyword can be used for this:

  ```
  MODULE = GNOME  PACKAGE = GNOME  PREFIX = gnome_

  char *
  gnome_util_user_shell()
  ```

  The above XS segment indicates that the function should appear to Perl as **`GNOME::util_user_shell`** rather than the more verbose and repetitive **`GNOME::gnome_util_user_shell`**.

- If a library provides a single and double precision interface consider using just the double precision interface unless there is a major performance penalty between the two. This is because all Perl floating point variables are already double precision and there is little point converting precision when transferring data between Perl and the library. If it is necessary to preserve the function names inside Perl (but as noted in a previous comment it may be better to adopt a more unified interface on the Perl side) it is possible to export both the single and double precisions names but only use the double precision function from the library. XS provides a way of doing 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 setup as an alias for `CalcDouble`. The `ix` variable is provided automatically and can be used to determine how the function was called.

2.7. Further Reading

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

perlmod
perlmodlib

Standard Perl manual pages on module creation

ExtUtils::MakeMaker

Description of `Makefile.PL`

perlxstut
perlxs

These are the standard XS tutorial and documentation that come with Perl itself. They cover everything about perl but they rapidly move onto advanced topics.

Notes

1. a makefile 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 Windows™ it will be necessary to install a version of **make**
2. on other systems it is still possible to use `DynaLoader` but the module must be statically linked into the perl binary by using **`make perl`** rather than just **`make`** for the second stage.
3. 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 is necessary
4. It may be necessary to rerun the makefile creation phase if the test program is created after the makefile has been created. This is because Perl adjusts the contents of the makefile depending upon what is present in the module distribution.
5. On Unix systems **`-lmylib`** refers to a file on disk called `libmylib.a` or `libmylib.so`. The former is a static library, the latter is a shared library that is loaded at runtime.

6. Modules for many Gnome libraries are already on CPAN
7. Gnome is usually installed into `/usr` on Linux but `/opt/gnome` on Solaris

Chapter 3. Advanced C

So far we have provided a simple introduction to the C programming language and used that knowledge to provide perl interfaces to simple C functions. Before we can progress on to a description of the perl internals, and the implementation of perl variables in particular, we need to introduce some more advanced C programming concepts. In this chapter we will show how to use arrays and data structures and how to deal with memory management.

3.1. Arrays

In Perl arrays are simply collections of scalars. Perl knows how many elements are in the array and allocates memory for new elements as required. In C, arrays are simply a contiguous block of memory that can only store variables of a specific type; an array of integers can only contain integers. Example 3-1 shows how you can create and access arrays of a known size in C along with the Perl equivalent. The use of square brackets is identical to the usage in Perl.

Example 3-1. Array handling in C

```
#include <stdio.h>                                          ❶
#include <stdlib.h>                                         ❶

int main () {
  int i;
  int iarr[5] = { 10,20,30,40,50 };                         ❷
  double darr[10];                                          ❸

  for (i=0; i<10; i++) {
    darr[i] = i/100.0;                                      ❹
  }

  for (i=0; i<5; i++) {
    printf("%d %d %f\n", i, iarr[i], darr[i]);              ❺
  }

  exit(0);
}

#!/usr/bin/perl

@iarr = ( 10, 20, 30, 40, 50 );

for (0..9) {
  $darr[$_] = $_/100.0;
}

for (0..4) {
```

```
  printf("%d %d %f\n", $_, $iarr[$_], $darr[$_]);
}
```

❶ Standard include files. These provide prototypes for `printf` and `exit`.

❷ Declare an integer array with 5 elements and then assign the elements using a comma-separated list. (Note the use of **`{ x, y, z }`** for Perl's **`(x, y, z)`**.)

❸ Create an array containing 10 double precision numbers. This array is created empty.

❹ Assign values to each of the elements in the double precision array.

❺ Print the contents of the first 5 elements of `iarr` and `darr`.

One complication for a C programmer is that you must keep track of the number of elements in the array and must allocate more memory if more elements are required (see Section 3.6.2). This is because the C compiler only allocates the memory that it knows you will need and not the memory you might need. To make things worse, C will not stop you "walking off the end" of the array, by assigning to an element that you have not allocated memory for. For instance, this is perfectly legal:

```
int array[10];

array[20] = 1234;
```

However, it will almost certainly cause annoying and occasionally difficult to detect problems, as you will be writing either into unallocated memory, possibly causing a segmentation fault, or, worse, into another variable's allocated storage.

3.2. Pointers

In the same way as in Perl all arguments in C are passed into subroutines by value so if you have a C function such as:

```
int findsum(int in1, int in2);
```

the subroutine will simply retrieve the values of `in1` and `in2` but will not be able to modify those variables directly. This is analogous to the Perl subroutine:

```
sub findsum {
  my $in1 = shift;
  my $in2 = shift;
```

```
  return ($in1 + $in2);
}
```

where the arguments are read from the stack and then used in the subroutine. In C there is no analogue of the variable aliasing trick available in perl (where `@_` can be modified in place) and in order to modify a variable the memory address of the variable must be passed to the function. Once the memory address is available it can be used to modify the value *pointed* to by this address. Given this definition it is not surprising to find that a variable that contains a memory address is called a *pointer*.

Pointers are associated with a particular variable type (much like a reference to a Perl hash is different from a reference to a Perl scalar) although, unlike in Perl, they can be converted to other types by using type-casting. This all means that a pointer to an integer is not the same as a pointer to a character *although you can force it to be treated as such*. In order to dereference a C pointer and modify the value of the variable the ***** notation is used and this is reflected in the declaration of a pointer:

```
char *str;
int  *x;
double *y;
```

This notation indicates that since, for example, `*x` is an integer then `x` itself must therefore be a pointer to an integer. An alternative reading is to think of `x` as a variable of type int * which is a pointer to an integer. A function that multiplies a variable by three could be implemented as:

```
void treble ( int * value ) {
    *value *= 3;
}
```

Here the input variable is declared as a pointer and the value in the variable is multiplied by three by dereferencing the pointer. This could be called as follows:

```
int number = 5;
treble( &number );
```

where we use an ampersand to indicate that we want to take the memory address of `number` and pass it to the function rather than simply passing in the value. The Perl equivalents would be:

```
$number = 5;
&treble( \$number );

sub treble {
  my $value = shift;
  $$value *= 3;
}
```

To summarise, `&variable` is used to retrieve the memory address and `*variable` is used to find the thing pointed to by the memory address stored in the variable.

There are some important differences between Perl references and C pointers. In Perl the reference can only be dereferenced and once dereferenced behaves as the original Perl data type. In C the memory address can be examined and modified. Modifying pointers can lead to errors though since bad things can happen if the pointer does not point to a valid location in memory. Additionally, a C pointer simply contains a single memory address therefore from that pointer it is not possible to determine whether it points to an array or a single variable.

A special type of pointer is a pointer to a void (`void *`). This simply indicates to the compiler that the contents are a pointer but we do not care what type of pointer it contains. This is very useful when designing data structures that are designed to be extensible or contain data of type that is not known until the program is run.

3.2.1. Pointers and Arrays

In Perl if you need to pass an array into a subroutine you usually do it by taking a reference to the array and storing it in a Perl scalar variable. Once the reference is stored in a scalar it must be dereferenced in order to retrieve the original variable:

```
&foo( \@bar, \%baz );

sub foo {
  my ($refArr, $refHash) = @_;
  $refArr->[3] = "foo";
  $refHash->{"bar"} = "foo";
}
```

In C the equivalent of this reference is a pointer. A C array is a block of contiguous memory and the start of the array is stored in a pointer. If the pointer is incremented by 1 (the compiler knows how many bytes to increment the pointer based on the pointer type) it will simply point to the next element of the array and inn fact a common way of stepping through an array in C is simply to increment the pointer. C arrays can be passed into subroutines simply by passing the pointer to the first element of the array into the subroutine; the subroutine then dereferences the pointer to obtain each of the elements of the array. In fact, the method we used above to retrieve element **`n`** from an array, **`array[n]`**, is functionally equivalent to incrementing the pointer to the start of the array by **`n`** and dereferencing the result: **`*(array+n)`**

3.3. Strings

As mentioned previously a string in C is treated as an array of characters. In general, a char is a single character and a char * is a pointer to an array of characters[1]. Since a C compiler does not know how many characters are contained in a particular piece of memory that is being referenced the convention in C is that a special character value is used to indicate the end of a string[2]. This string termination character is character `\0` (the character with value 0 in the character set [**`chr(0)`** in Perl] not character zero

[**ord(0)** in Perl]) and is known as the NUL terminator. Example 3-2 shows a simple example of string manipulation using a pointer and a Perl translation. The example shows that in Perl a dereferenced reference is no different to the original (the example would be identical if `$$b` was replaced with `$a` throughout) and that a string in Perl is a single entity whereas in C it is a group of characters.

Example 3-2. String manipulation

```
$a = "hello";
$b = \$a;
substr($$b, 0, 1) = 'm';
substr($$b, 3, 1) = 'd';
$a = substr($$b, 0,4);
```

Example of how a string can be manipulated a character at a time using pointers along with a somewhat contrived Perl equivalent since perl manipulates strings but C manipulates characters.

3.3.1. Arrays of strings

Unlike numeric arrays where multi-dimensional arrays are a contiguous block of memory (since a single number is always represented by a fixed number of bytes), arrays of strings are arrays of pointers to strings. The strings themselves can be in completely unrelated areas of memory. A pointer to an array of pointers to chracters is represented by the type char**. If the variable `x` is a char**, `*x` is a pointer to a string and **x is the single character at the start of the string. This is represented graphically in Example 3-3.

Example 3-3. Arrays of strings

The memory organization of an array of strings. On the left we have a simple pointer. This pointer (a char**) points to an array of pointers (shown in the middle). These pointers are of type char* and point to the first character in each string. We use the NULL at the end of the array to tell us where the end is.

As a simple example, of using a char** we can make use of the Unix environment. The environment is available to programmers as an array of strings. For example, if we wanted to write a small program to print our current environment (similar to the **printenv** shell command) we could do something like this:

```
#include <stdio.h>

extern char** environ;                                  ❶

int main () {
  int i;
  char **array = environ;                               ❷

  while (*array) {                                      ❸
```

```
    printf("%s\n", *array);
    array++;
  }

  return(0);
}
```

❶ `environ` is a special variable provided by the system and populated when the process starts. The extern declaration tells the compiler that we are using a variable declared in some other file.

❷ Here we declare a new char** and copy the pointer from the environment array. We do this so that we can manipulate the pointer without affecting the base pointer.

❸ This while loop continues until `*array` evaluates to false. Since `*array` is itself a pointer this only happens when we are pointing to a null. Each time round the loop we print out the string associated with the current pointer value and then increment the pointer so that we move to the next element in the array. When we reach the end of the array we hit the null and terminate the loop.

3.4. Structures

In many cases it is desirable to group related data into a single unit that can be passed easily into subroutines. In perl this is achieved by using a hash or an array which can themselves contain references to any of the standard perl data types and can even be blessed into classes and treated as objects. In C arrays can only contain variables of the same type[3] and an alternative method is provided for grouping data into structures that can contain named elements similar to a Perl hash. These structures, declared using the `struct` keyword, can be given an arbitrary type and can also include any other variable type:

```
struct person {
   double income; /* A double precision number */
   int    age;    /* An integer */
   char*  name;   /* Pointer to a character string */
};

struct survey {
   struct person* respondent; /* Pointer to a structure of type person */
   float length;       /* Floating point number (single precision) */
   int* answers;       /* Pointer to an array of integers */
};
```

Once a structure has been declared variables can be declared of that type in the normal way:

```
struct person mystruct; /* mystruct is now a variable of type person */
struct survey* mypoint; /* mypoint is now a pointer to a struct of type survey */
```

A common trick to save typing is to instruct the compiler to use an alias rather than having to type **`struct person`** all the time. The `typedef` command allows any variable type to be aliased to another:

```
typedef int myint;  /* myint is now identical to an int declaration */
typedef struct person PERSON;  /* PERSON is now an alias for struct person */
struct person mystruct; /* mystruct declared as a struct person */
PERSON mystruct;        /* Same as previous line */
```

This technique is used a lot in the Perl source code where you very rarely see an explicit struct. All struct accesses are done with aliases (and even more commonly done with C pre-processor macros).

Accessing the members of the structure depends on whether you have a pointer to the structure or have declared the structure variable directly. The following example shows C struct accessors with a corresponding perl hash accessor:

```
mystruct.age = 5; /* Set member 'age' to 5 */
$mystruct{'age'} = 5;  # perl

mypoint->length = 5.234; /* Set member y by dereferencing pointer */
$mypoint->{'length'} = 5.234; # Perl dereference

(*mypoint).length = 5.234; /* Same as previous line */
$$mypoint{'length'} = 5.234; # Alternative form of dereferencing
```

The main difference between a struct and a hash is that all the members of a struct must be declared beforehand and no extra members can be created as the program executes. This is because a struct, like an array, is a contiguous block of memory of a fixed size. Even if a particular element of a struct is not initialised it will still take up the same amount of memory as a fully initialised struct. Our struct person declared above consists of a double, an int and a pointer to a string. On a normal 32-bit system this will most likely take up 8+4+4=16 bytes of memory; the standard `sizeof` can be used to determine how much memory the structure actually uses (in some cases a compiler will pad the structure with extra bytes to make it a convenient size for the underlying architecture). Structures must be well-defined like this so that the compiler can decide on how much memory to allocate to each one. This is all clearly different from a perl hash where keys can be added at any time and the memory requirement is not fixed. Details on how hashes are implemented in C to overcome any restriction of this kind can be found in Section 4.4. Additionally as a structure is represented by a single contiguous block of memory (just like arrays) it is possible to simply step the required number of bytes into a structure to extract out information but this is not recommended and can lead to confusion and difficulty in porting the code to other platforms; always use the standard accessor techniques.

In a similar manner to arrays, it is possible to initialise a structure in one statement:

```
struct example {
  double a;
  float b;
  int c;
};
```

```
struct example eg = { 52.8 , 65.4, 40 };
```

This provides a useful shorthand way of configuring structures without having to address each member by name. The curly brackets effectively packing the data into a single block of memory and is exactly the same as using the Perl `pack`:

```
$packed = pack("dfi", 52.8,65.4,40);
```

3.5. File I/O

In C, just as in Perl[4], there are two approaches to handling input from and output to external files[5]:

- Stream-based I/O provides a high level interface, specifically providing buffering facilities and the ability to move forwards and backwards within a file. Buffered I/O is important for performance reasons (when writing to real hardware it is usually more efficient to store up writes to a disk into large chunks rather than sending a byte at a time). By default perl filehandles use streams and the buffering can be turned on or off using the special variable `$|` (the default is to turn on buffering leading to much confusion from people the first time they write a CGI script!). In C a stream is represented by a `FILE*` (a pointer to a FILE). This is an opaque structure since you never look inside the structure - that is left to the low level implementation in the C library itself. To open a stream you can use `fopen`:

  ```
  FILE *fopen(const char *path, const char *mode);
  ```

  and to print to a file you can use `fprintf`

  ```
  int fprintf(FILE *stream, const char *format, ...);
  ```

  To close it you use, predictably, `fclose`:

  ```
  int fclose(FILE *stream);
  ```

  These are very similar to the corresponding perl routines, `open`, `printf` and `close`.

- A lower level approach to I/O is to use *file descriptors*. These are simple integers (you can get the file descriptor from a perl filehandle using the `fileno` routine). File descriptors can be used for non-buffered I/O and are especially useful for socket communication. The C functions `open`, `close`, `read` and `write` are direct equivalents of the Perl functions `sysopen`, `close`, `sysread` and

`syswrite`. Just as in Perl, you should not mix stream-based I/O with file descriptor I/O on the same file handle.

In general all of the perl file I/O operations have analogues in the C library either as stream-based functions or functions using file descriptors (the `perlfunc` documentation is careful to distringuish between the two).

3.6. Memory Management

So far we have explained the different variable types and always made sure that our program has specified the amount of memory required to use them. In Example 3-1 we specified the number of elements for each array. The memory for these variables is allocated by the compiler which works out the total amount of memory required by the program. Unfortunately, sometimes you can not know at compile time how much memory you program will need. For example, if you are writing some image processing software you may not know how big an image you will need to process. One solution, required if you use a language such as Fortran77[6], is to make an inspired guess of the maxium image size you wish to process and hard-wire the dimensions into your program. This will work but has two problems:

- If you need to process larger images you will need to recompile your program (presumably this is easy because you have specified the dimensions in a header file?)
- Even if you are processing small images the program will require the same amount of memory as it uses for large images.

Clearly an alternative approach is required. What we really want to do is to determine the image size that we are intending to process and then allocate that memory when the program is running.

3.6.1. Allocating memory at runtime

The main C function for *dynamically* allocating memory is `malloc`:

```
void * malloc(size_t size);
```

The argument specifies the number of bytes required and the function returns a pointer which can be type-cast to the correct type. Recall that if you declare a pointer variable, you have to make sure that it points to some allocated memory. So far we have done this by obtaining the pointer from some other variable. For example:

```
char * pointer_to_char;
char a_string[4];
int an_integer;
int * pointer_to_int;
```

```
pointer_to_char = a_string;
pointer_to_int = &an_integer;
```

With dynamic memory allocation we can do this (the prototype for `malloc` can be found in `stdlib.h`:

```
char * pointer_to_char;

pointer_to_char = (char *)malloc(4);
```

Here we have requested 4 bytes of memory and we have stored the pointer. It is important to realise here that the memory has been allocated but not necessarily initialised (the contents will be undefined). You are responsible for storing information in that memory.

The downside of dynamic memory allocation in C is that the programmer is responsible for giving the memory back when it is no longer required. If we keep on requesting memory but never give it back we will rapidly run out of resources. This is known as a *memory leak*. C provides the `free` to allow you to return memory to the system:

```
void free(void * ptr);
```

This function takes a single pointer argument and will free the memory at that address. This memory must have been allocated by a call to `malloc` (or the related functions `calloc` and `realloc`) and care should be taken that it is not called twice with the same pointer value (strange things may happen since the system may have already given that memory to something else!).

Memory management is one of the hardest things to get right in C. With a large program containing many dynamic memory allocations it is very hard to guarantee that the memory will be freed correctly in all cases. One of the key advantages of Perl, Java and others is that they handle all the memory allocation and freeing, allowing the programmer to focus on core functionality of the program.

3.6.2. Altering the size of memory

Occasionally you may realise that the memory you requested was not enough for the task in hand: maybe you want to extend an array so that it can hold more elements, maybe you want to extend a string. The *obvious* approach is to allocate a new string of the correct size, copy the contents from the first string and then free the memory associated with it. This is painful but luckily the standard C library comes with a function that will do this for you:

```
void * realloc(void * ptr, size_t bytes);
```

This function will resize the available memory pointed to by `ptr`. The original contents of the memory will be retained and any new memory will be unintialized. If `bytes` is less than the previous size then the additional memory will be freed. `realloc` returns a new memory address which will not necessarily be the same as that stored in `ptr`.

3.6.3. Manipulating Memory

Now that you have your memory allocated, what can you do with it? If you want to copy it to another variable (for example to copy the contents of a structure before it is reused) you can use `memcpy` (use `memmove` if the memory destination lies within the chunk of data that is being copied).

```
#include <stdlib.h>                                          ❶
#include <stdio.h>                                           ❶
#include <string.h>                                          ❶

typedef struct {                                             ❷
   double income; /* A double precision number */
   int    age;    /* An integer */
   char*  name;   /* Pointer to a character string */
} person;

int main () {                                                ❸

  char name[6] = "fred";                                     ❹
  person someone = { 20000.0, 5, name };                     ❹
  person *other;                                             ❹

  other = (person *)malloc(sizeof(person));                  ❺
  memcpy(other, &someone, sizeof(person));                   ❻

  other->income = 0.0;                                       ❼
  printf("Someone: %d, %6.0f, %s\n", someone.age, someone.income,❽
 someone.name);

  strcpy(someone.name, "wilma");                             ❾
  printf("Other:   %d, %6.0f, %s\n", other->age, other->income,(10)
 other->name);

  free(other);                                               (11)
  return(0);
}
```

❶ These include files declare the prototypes for `malloc`, `printf`, `memcpy` and `strcpy`

❷ Here we create an anonymous structure and simultaneously typedef it to a person.

❸ This is the standard C main routine but here we do not declare any arguments since we are not interested in them.

❹ Variable declarations. We allocate a string, initialise a person structure (that includes the previously declared string) and declare a pointer to a person.

❺ Allocate memory for a new structure. We determine the number of bytes required by using `sizeof`.

❻ Use `memcpy` to copy the contents of `someone` to the memory indicated by `other`. We must supply the size of the structure in bytes.

❼ Set the income in `other` to zero.

❽ Print out the contents of the original structure.

❾ Set the name in the original structure to "wilma"

(10) Print out the contents of the copy.

(11) Free the dynamically allocated memory.

If we run the above program we get the following output:

```
% gcc -Wall memcpy.c -o memcpy
% ./memcpy
Someone: 5,  20000, fred
Other:   5,      0, wilma
```

So, when we modified the income of `other` it did not affect the contents of `someone` but when we modified the name of `someone` it did affect `other`. What is going on here? The answer lies in the in the struct declaration. When the structure is copied, everything is copied exactly to the destination. The first two entries are simply numbers and are copied as such. If they are modified the bytes change as we would expect without affecting anything else. The third member is simply a pointer. This means that the pointer is identical in both structures and if that memory is modified both structures "see" the change.

If you want to initialise an array with a default value you can use the `memset` (or you can allocate the array using the `calloc` which allocates the memory and then fills it with zeroes)

```
#include <stdlib.h>                                    ❶
#include <stdio.h>                                     ❶
#include <string.h>                                    ❶

#define NELEM 10                                       ❷

int main () {                                          ❸

  int i;
  int * array;                                         ❹
  array = malloc(sizeof(int)*NELEM);                   ❺
  memset(array, 0, sizeof(int)*NELEM);                 ❻

  for (i=0; i<NELEM; i++) {
    printf("Index %d element %d\n",i, array[i]);       ❼
  }
```

```
  free(array);                                                    ❽
  return(0);
}
```

❶ These include files declare the prototypes for `malloc`, `printf` and `memset` respectively.

❷ We declare the size of the array in a single place so that we don't need to rely on a bare number in the code. Remember that C arrays do not know their size.

❸ As with the previous example this is the standard C main routine but we do not declare any arguments since we are not interested in them.

❹ Declare `array` as a pointer to an integer.

❺ Allocate `NELEM` integers. We multiply the number of elements by the number of bytes in each integer to determine the total number of bytes required.

❻ Set each byte in the new array to zero.

❼ Loop through each element of the array, printing it to the screen.

❽ Free the memory.

3.6.4. Memory manipulation and Perl

* *This is discussed to some extent in the chapter on perl API and we could get away with moving this section straight into that chapter. We want to describe these specially here because they are the most common wrappers that are used by XS. They are not the only ones and we may feel a need to include the others listed in `perlclib` as well.*

Having said all this, in the perl source code you will never use any of the functions for memory allocation and manipulation that we have described above. In the interests of cross-platform portability and debugging support Perl uses private definitions of these functions by providing C pre-processor macros. The `New` may eventually translate to a `malloc` but it doesn't have to. If we are attempting to check for memory leaks we may want to redefine `New` so that it keeps track of the allocations and corresponding frees and a macro makes this extremely simple. Table 3-1 contains a list of the standard C library functions and the Perl equivalents. In the table `t` indicates a variable type, `p` indicates a pointer, `n` is a number and `s` is a string. `dst`, `src` and `id` indicate destination pointer, source pointer and identification tag respectively. The `id` could be used to track memory leaks although it is not currently used in the Perl source.

Table 3-1. Perl macros for memory manipulation

Perl source	C library
`New(id, p, n, t);`	`t* p = malloc(n);`
`Newz(id, p, n, t);`	`t* p = calloc(n, s);`

`Renew(p, n, t);`	`p = realloc(p, n);`
`Safefree(p);`	`free(p);`
`Copy(src, dst, n, t);`	`memcpy(dst, src, n);`
`StructCopy(src, dst, t);`	`memcpy(dst, src, n);` (structures only)
`Move(src, dst, n, t);`	`memmove(dst, src, n);`

3.7. Summary

In this chapter we have learnt the following:

- Arguments are passed into functions by value. In order to modify a variable in a function a pointer to the variable must be used.
- Arrays in C are simply blocks of memory that can be addressed by a pointer. They can not be re-sized without allocating more memory and they do not know how long they are.
- C has no native hashes, only fixed size structures.
- C will not automatically allocate more memory as you need it. You have to explicitly ask for more memory and you have to make sure you give it back to the system when it is no longer required.

3.8. Further reading

There are many books available on the C programming language and we can not recommend all of them. The standard reference works are Kernighan & Ritchie's "The C programming language" and Harbison & Steele. [fill in title]. Also, all the library functions described here have corresponding manual pages on most systems.

Finally, the Perl wrappers for standard C functions are described in the `perlclib` documentation that comes with perl.

Notes

1. A character is not necessarily a single byte. This is true in ASCII but not in UNICODE

2. This leads to problems when linking Perl via C to languages that do not use this particular convention. Fortran is a popular example where the compiler does know how long each string is and therefore does not need a special string termination character
3. although in principle you could have an array of pointers to items of different type in practice it is difficult to keep track of which element points to which data type.
4. simply because Perl provides interfaces to both types of I/O in the standard C library
5. This includes sockets and devices. On Unix all devices are treated as files.
6. although most modern implementations of Fortran77 ignore the restriction

Chapter 4. Perl’s Variable Types

This chapter will explain how Perl variables (scalars, arrays, hashes and globs) are represented inside Perl. Starting from the various scalar types (`$x` in Perl) it will then continue to discuss magic (for example, ties), the more complex data types and the organisation of namespaces and lexical (“my”) variables.

This will be the first detailed look inside Perl and will use knowledge of the types of variables used in C and how C data structures are implemented. This chapter assumes no more knowledge of C than that found in the previous chapters.

4.1. Scalar variables

A Perl variable is much cleverer than a simple C variable. Perl knows how many characters the variable needs, how to convert it from a number to a string and how many other variables know about it (so that Perl can tidy up after itself). This is achieved by using a C data structure (a struct) rather than a simple variable or array. As explained in the previous chapter a C struct is a block of memory that can contain any number of variables of any type. Each entry in the struct can be accessed by name and is functionally equivalent to a simplified Perl hash.

The simplest of all perl variable types is a scalar (e.g. `$xyz`) and this is represented inside Perl as a C structure of type SV (SV stands for *Scalar Value*)[1].

4.1.1. SvNULL

The basic implementation of an SV from which all perl variables are derived is the SvNULL structure:

* *Would like to show these and similar diagrams in a more graphical way showing the data structure in a more approachable way. That is why I also give a "perl" equivalent. Would be annotated in the way indicated to show sv_any, sv_refcnt, sv_flags and explaining that a U32 is a int of at least 4 bytes*

```
struct sv {                                    ❶
   void* sv_any;                               ❷
   U32   sv_refcnt;                            ❸
   U32   sv_flags;                             ❸
}
```

❶ This creates a new structure named `sv`

❷ This is simply a pointer of any allowed type.

❸ This is an unsigned integer that must be at least 4 bytes (32 bits) long.

Using perl hash syntax this would become:

```
$sv = {
        sv_any => undef,
        sv_refcnt => 1,
        sv_flags  => 0
      };
```

The actual fields (`sv_any`, `sv_refcnt` and `sv_flags`) can be accessed using C macros (defined in the `sv.h` include file) `SvANY`, `SvREFCNT` and `SvFLAGS` (these also match the output provided by the `Devel::Peek` module described later). From now on, the structure fields will be named after the macro name (without the leading Sv string) rather than the actual name used to define the structure since all programs written using the internals go through the provided macros rather than directly to the structure.

The `ANY` field is used to point to an additional structure that contains the specific state of the variable and can be changed depending on whether an integer (IV), double precision floating point number (NV) or character string (PV; standing for pointer value rather than *string value* since SV is already taken and a string in C is defined by using a pointer to the memory location of the first character). Variables are implemented in this way so that the basic structure of the variable is well defined such that all Perl variables, however complex the implementation, still have the same toplevel organization. Whenever it is necessary to store new information in the SV the structure pointed to by the `ANY` field is changed to accomodate the new data (it is *upgraded* to the new form). It is not possible to *downgrade* an SV to a type containing less information since that would throw information away. The different types are discussed later on in this chapter.

For an SvNULL structure the `ANY` field does not point to anything (it contains a `NULL` pointer)[2] so this structure represents a scalar with a value of `undef`.

The `REFCNT` field contains the current reference count for the variable. Perl decides whether the memory associated with a variable can be recycled on the basis of reference counting. When a variable is created it has a reference count of 1. Whenever a reference of that variable is stored somewhere the reference count is increased by 1 and whenever a reference is no longer required the reference count is decreased by 1. If the reference count goes to zero the variable can no longer be used by Perl and the memory associated with it is freed.

The `FLAGS` field contains bit flags that can be used to determine the behaviour of certain fields and the current state of the variable (for example, whether the variable is a package variable or a lexical variable).

4.1.2. Looking inside - `Devel::Peek`

When you are developing Perl extensions (or are simply interested in what is happening to a variable) it is very useful to be able to examine the internal structure of a Perl variable from a Perl program. The `Devel::Peek` module is available as part of the standard perl distribution and provides the functionality for doing just that. It can be used to list the current state of a variable in detail.

In the following example, the Perl code is on the left and the corresponding output from each line is on the right:

```
use Devel::Peek;
$a = undef;
Dump $a;                        SV = NULL(0x0) at 0x80f9bf0 ❶
                                  REFCNT = 1               ❷
                                  FLAGS = ()               ❸

$b = \$a;
Dump $a;                        SV = NULL(0x0) at 0x80f9bf0
                                  REFCNT = 2               ❹
                                  FLAGS = ()

print $b;                       SCALAR(0x80f9bf0)          ❺
```

❶ `Devel::Peek` shows us that we have an SVNULL structure, and it tells us the memory address of that structure - `0x80f9bf0` in this particular instance; expect your output to show a different location. The address in brackets (`0x0`) tells us where the `ANY` field in the SV's structure is pointing - in the case of an SVNULL, this is zero, but in more complicated SVs, it will point to another structure in memory.

❷ Initially, we have one reference to the SV.

❸ The SV's flags are empty.

❹ When we create a new reference to the SV, its reference count increases by one.

❺ When the reference is printed (the contents of `$b`) note that the stringified reference that Perl generates includes the memory address.

4.1.3. Flags

Much of the state information for an SV is contained in the `FLAGS` field and before proceeding further with the more complex SV structures we will explain how the flag system works. All computer data are stored in terms of binary digits (bits). A particular bit can either be set (a value of `1`) or unset (`0`) and the state of a bit can be checked by comparing it with a bit mask using binary logic (a bitwise `AND` can be used to determine if a bit is set, a bitwise `OR` can be used to set a bit). The following example uses a 4-bit number and compares it with a bitmask to determine whether the specific flag is set (that is, both the bit in the number to be checked and the bit in the bitmask are set):

```
               Binary     Decimal
Bit number     3 2 1 0

FLAG           0 1 1 0      6
BITMASK        0 0 1 0      2
               -------
AND            0 0 1 0      2
```

Here the flag has a decimal value of 6 (`0b0110` in binary) and when a bitwise `AND` operation is performed with the bit mask the result is non-zero and indicates that the bit that is set in the bit mask is also set in the flag. If the result of the operation was zero it would mean that the flag was not set. In the above example 4 independent states can be stored since we are using a 4 bit number. In \perl\ the `FLAGS` field is implemented using a 32-bit integer and so it is possible to record 32 different states. The actual size of the flag variable or the bit number associated with each of the states is completely irrelevant as those values are set internally in the C include values. All that really matters is that Perl provides C routines that can be used to query an SV for a particular state. For example, in order to see whether an SV is readonly the `SvREADONLY` macro can be used:

```
  if ( SvREADONLY( sv ) )
      printf "SV is readonly\n";
```

The following example shows a Perl implementation of the READONLY flag used above:

```
use 5.006;  # binary 0b0010 notation needs perl 5.6
use constant SVf_READONLY => 0b0010; # Set the mask bit

sub SvREADONLY { $_[0] & SVf_READONLY } # Test the mask bit
sub SvREADONLY_on( $_[0] |= SVf_READONLY } # Set the READONLY bit
sub SvREADONLY_off( $_[0] &= ~SVf_READONLY } # Unset READONLY bit

# Set the flag
$flag = 0;
SvREADONLY_on( $flag );

# Test for the flag
print "Flag is readonly\n" if SvREADONLY( $flag );
```

The important point is that in the above example the programmer only ever uses the `SvREADONLY` subroutines and never needs to use the `SVf_READONLY` value directly (or even care what its value is).

When the `Devel::Peek` module is used it lists all the flags that are currently set in the variable:

```
% perl -MDevel::Peek -e 'Dump( my $a )'
SV = NULL(0x0) at 0x80e58c8
  REFCNT = 1
  FLAGS = (PADBUSY,PADMY)
```

Here the PADMY flag is set indicating that `$a` is a lexical variable (The PADBUSY flag is set for the same reason; see Section 4.7). The important flags will be discussed in this chapter as the relevant variable types are discussed.

4.1.4. SvRV - references

The simplest possible perl variable that actually contains data is the SvRV subtype and it is used to contain references to other SV's. An SvRV is simply an SV where the `ANY` field points to a simple structure (named `xrv`) that contains a single field that is a pointer to another SV (an SV*):

```
struct xrv {
    SV *        xrv_rv;         /* pointer to another SV */
};
```

Represented as a perl structure **`$b=\$a`** would be

```
  $b = {
          ANY    => { RV => \$a },
          REFCNT => 1,
          FLAGS  => ROK
       }
```

or using `Devel::Peek`:

```
SV = RV(0x80fbabc) at 0x80f9c3c                          ❶
  REFCNT = 1                                             ❷
  FLAGS = (ROK)                                          ❸
  RV = 0x80ef888
  SV = NULL(0x0) at 0x80ef888                            ❹
    REFCNT = 2                                           ❺
    FLAGS = ()
```

❶ This SV is of type SvRV

❷ This reference has a ref count of 1 - that is, `$b`

❸ The ROK flag (*Reference OK*) bit is set to true to indicate that the reference is valid (if ROK is false the variable contains the undefined value).

❹ This line tells us about the SV that is being referred to, the SV stored in `$a`.

❺ We see that this SV has two references: the value itself, in `$a`, and the reference to it, in `$b`.

4.1.5. SvPV - string values

Perl variables that contain just a string representation are type SvPV (they simply contain a PV). They are represented by a SV with the `ANY` field pointing to a structure that contains a pointer to a string (a char * that is named `PVX`) and two length fields (the `CUR` and `LEN` fields).[3]

```
    ANY  =>  PVX  =>        hello world\0
```

```
          CUR  =>        <--------->
          LEN  =>        <----------->
  REFCNT
  FLAGS => POK | pPOK    <--- POINTER OK flag
```

The PVX field contains a pointer to the start of the string representation of the SV. The CUR field is an integer containing the length of the perl string and the LEN field is an integer containing the actual number of bytes allocated to the string. Additionally, the byte at position **(PVX + CUR)** must be a "\0" (C uses a NUL byte to indicate the end of a string) so that other C functions that receive this string will handle it correctly. This means that LEN must be at least 1 more than the value of CUR. Perl's memory management is such that for efficiency it will not deallocate memory for the string once it has been allocated if the string is made smaller. It is much more efficient simply to change the value of CUR than it is to free the unused memory when a string becomes shorter.

```
use Devel::Peek;

$a = "hello world";
Dump $a;                SV = PV(0x80e5b04) at 0x80f9d98
                          REFCNT = 1
                          FLAGS = (POK,pPOK)
                          PV = 0x80e9660 "hello world"\0
                          CUR = 11
                          LEN = 12

$a = "hello";           SV = PV(0x80e5b04) at 0x80f9d98
Dump $a;                  REFCNT = 1
                          FLAGS = (POK,pPOK)
                          PV = 0x80e9660 "hello"\0
                          CUR = 5
                          LEN = 12
```

The POK flag indicates that the PV stored in the variable is valid and can be used. The pPOK flag is related to this but is an internal flag to indicate to the Magic system (see Section 4.2) that the PV is valid.

4.1.6. SvPVIV - integers

In C, it is not possible to store a number or a string in a variable interchangably. Perl overcomes this restriction by using a data structure that contains both a string part and an integer part, using the flags to indicate which part of the structure contains valid data. The name SvPVIV indicates that the structure contains a string (PV) and an integer (IV) and is simply an SvPV with an extra integer field.

```
  ANY   => { PVX  =>  ”
             CUR  =>  0,
             LEN  =>  0,
             IVX  =>  52,
           }
  FLAGS => IOK | pIOK | IsUV   <--- Integer OK and is an unsigned value
```

This structure introduces three flags. The IOK and pIOK flags indicate that the `IVX` field is valid (in the same way that POK and pPOK indicate that the string part is valid) and the IsUV flag indicates that the integer part is unsigned (a UV) rather than signed (an IV). This is useful in cases where a large positive integer is required (such as inside loops) since a UV has twice the positive range of a signed integer and is the default state when a new variable is created that contains a positive integer.

When a string value is requested (using the `SvPV` function) the integer is converted to a string representation and stored in the `PVX` field and the POK and pPOK flags are set to true to prevent the conversion happening every time a string is requested.

```
use Devel::Peek;

$a = 5;
Dump $a;                        SV = IV(0x80f0b28) at 0x80f9d0c
                                  REFCNT = 1
                                  FLAGS = (IOK,pIOK,IsUV)       ❶
                                  UV = 5                    ❶
# string comparison
print "yes" if $a eq "hello";
Dump $a;                        SV = PVIV(0x80e5f50) at 0x80f9d0c
                                  REFCNT = 1
                                  FLAGS = (IOK,POK,pIOK,pPOK,IsUV) ❷
                                  UV = 5
                                  PV = 0x80e9660 "5"\0             ❷
                                  CUR = 1
# Copy in a new string            LEN = 2
$a = "hello";
Dump $a;                        SV = PVIV(0x80e5f50) at 0x80f9d0c
                                  REFCNT = 1
                                  FLAGS = (POK,pPOK)                  ❸
                                  IV = 5                              ❸
                                  PV = 0x80e9660 "hello"\0            ❸
                                  CUR = 5
                                  LEN = 6
```

❶ Initially the SV simply contains a UV and a flag indicating that the integer part of the SV is okay.

❷ The string comparison forces the SV to be stringified. This results in an upgrade to a PVIV, the POK flag is set to true in addition to the IOK flag and a string representation of the number stored in the PV slot.

❸ The string part is modified so the integer part is now invalid. The IOK flag is unset but the IV retains its value.

4.1.7. SvPVNV - floating point numbers

For the same reason that a C variable can not contain a string and an integer, it can not contain a floating point value either. This is overcome simply by adding a floating point field to an SvPVIV.

```
    ANY   => { PVX  =>  ”
               CUR  =>  0,
               LEN  =>  0,
               IVX  =>  0,
               NVX  =>  54.5e20
             }
    FLAGS => NOK | pNOK       <----- Number OK
```

As for the other types, previous settings for the string and integer parts are retained as the variable evolves, even if they are no longer valid.

```
use Devel::Peek;
$a = "hello world";
$a = 5;
$a += 0.5;
Dump $a;
```

gives:

```
SV = PVNV(0x80e65c0) at 0x80f9c00
 REFCNT = 1
 FLAGS = (NOK,pNOK)
 IV = 5
 NV = 5.5
 PV = 0x80e9660 "hello world"\0
 CUR = 11
 LEN = 12
```

where the IV and PV parts retain their old values but the NV part is the only value that is currently valid (as shown by the flags).

4.1.8. SvOOK - offset strings

In order to improve the speed of character removal from the front of a string, a special flag is provided (OOK - Offset OK) that allows the IVX part of the SV to be used to represent an offset in the string rather than being an integer representation of the string.

```
   ANY =>   PVX =>  ..llo world\0
            CUR =>    <------->
            LEN =>    <--------->
            IVX =>  <>
   FLAGS => POK | pPOK | OOK  <---- Pointer OK and Offset OK
```

It is not possible for the IOK flag and the OOK flags to be set at the same time as the IVX can not be both an offset and a valid number.

The use of this flag is best demonstrated by example:

Example 4-1. SvOOK example

```
% perl -MDevel::Peek -e '$a="Hello world"; Dump($a); $a=~s/..//; Dump($a)'
SV = PV(0x80f89ac) at 0x8109b94
  REFCNT = 1
  FLAGS = (POK,pPOK)                                    ❶
  PV = 0x8109f88 "Hello world"\0                        ❷
  CUR = 11                                              ❸
  LEN = 12                                              ❸
SV = PVIV(0x80f8df8) at 0x8109b94
  REFCNT = 1
  FLAGS = (POK,OOK,pPOK)                                ❹
  IV = 2  (OFFSET)                                      ❺
  PV = 0x8109f8a ( "He" . ) "llo world"\0               ❻
  CUR = 9                                               ❼
  LEN = 10                                              ❼
```

❶ Standard flags for a PV

❷ The string stored in the PV

❸ The length of the string and the size of the buffer allocated for storage.

❹ After processing the PV now has an additional flag (OOK) indicating that offsetting is in effect.

❺ This indiciates the real "start" of the string. `Devel::Peek` indicates that this is an offset.

❻ This shows the string split into two parts: the piece that is ignored at the start and the current value.

❼ The length information is now relative to the offset

4.2. Magic Variables - SvPVMG

In Perl a magic variable is one in which extra functions are invoked when the variable is accessed rather than simply retrieving the PV, IV or NV part of the SV. Examples are tied variables where the `FETCH` and `STORE` routines (plus others) are supplied by the programmer, the `%SIG` hash where a signal handler is set on assignment or the `$!` variable where the C level `errno` variable is read directly. Additionally, objects make use of magic when they are blessed into a class.

```
ANY =>  PVX
        CUR
        LEN
        IVX
        NVX
        MAGIC => { # MAGIC
                   moremagic => magic
                   virtual    => virtual table of functions
                   private
                   type
                   flags
                   obj
                   ptr
                   len
                 }
        STASH => { Foo::Bar => {} }
```

An SvPVMG magic variables is just like an SvPVNV variable except that two extra fields are present (the structure attached to the `ANY` field is of type xpvmg). The `MAGIC` field points to an additional structure of type magic and the `STASH` field points to a namespace symbol table relating to the object (stashes are described later in Section 4.6). When the `STASH` field is set (that is, the SV is blessed into a class) the OBJECT flag is set.

```
use Devel::Peek;
$a="bar";
$obj = bless( \$a, "Foo::Bar");
Dump $obj;
```

```
SV = RV(0x80fb704) at 0x80f9d44
  REFCNT = 1
  FLAGS = (ROK)
  RV = 0x80f9d98
  SV = PVMG(0x8102450) at 0x80f9d98
    REFCNT = 2
    FLAGS = (OBJECT,POK,pPOK)
    IV = 0
    NV = 0
    PV = 0x80e9660 "bar"\0
    CUR = 3
    LEN = 4
    STASH = 0x80f9ce4   "Foo::Bar"
```

The important entries in the magic structure (defined in `mg.h`) are the following:

moremagic

: This is simply a pointer to a linked list of additional MAGIC structures. Multiple MAGIC can be associated with each variable.

virtual

This is a pointer to an array of functions. Functions can be present for retrieving the value ("`get`"), setting the value ("`set`"), determining the length of the value ("`len`"), clearing the variable ("`clear`") and freeing the memory associated with the variable ("`free`"). In perl this is equivalent to

```
$virtual = {
      "get" => \&get,
      "set" => \&set,
      "len" => \&len,
      "clear" => \&clear,
      "free" => \&free
      };
```

obj

This is some data that can contain a pointer to anything that is important for the type of magic being implemented. For a tie this will be an SV of the tied object.

type

This is a single character denoting the type of magic implemented. A value of 'P' indicates that the magic is a tied array or hash and 'q' indicates a tied scalar or filehandle. A value of '~' or 'U' indicates that the functions in the virtual table have been supplied by the programmer. An extensive list of the different types can be found in the `perlguts` documentation.

At least one of the MAGIC flags will be set. The important flags are GMAGICAL (the SV contains a magic get or len method), SMAGICAL (the SV contains a magic set method) or RMAGICAL (the SV contains some other form of *random* magic).

* *[LARGE DIAGRAM SHOWING MAGIC TIE + pPOK by calling Dump twice]*

```
use Devel::Peek;
tie $a, 'Tie::Foo';
Dump $a;
$b = $a;
Dump $a;
exit;

package Tie::Foo;
sub TIESCALAR { my $obj="foo"; return bless(\$obj,"Tie::Foo"); };
sub FETCH { ${ $_[0] }++ }
```

```
SV = PVMG(0x8102470) at 0x80f9d98
  REFCNT = 1
  FLAGS = (GMG,SMG,RMG)
  IV = 0
  NV = 0
  PV = 0


  MAGIC = 0x80ebbf8
```

```
SV = PVMG(0x8102470) at 0x80f9d98
 REFCNT = 1
         FLAGS = (GMG,SMG,RMG,pPOK)
 IV = 0
 NV = 0
 PV = 0x80f0f80 "foo"\0
 CUR = 3
 LEN = 4
  MAGIC = 0x80ebbf8
```

```
MG_VIRTUAL = &PL_vtbl_packelem
MG_TYPE = 'q'
MG_FLAGS = 0x02
  REFCOUNTED
MG_OBJ = 0x80e58c8
SV = RV(0x80fb6f8) at 0x80e58c8
  REFCNT = 1
  FLAGS = (ROK)
  RV = 0x8103a00
  SV = PVMG(0x8102450) at 0x8103a00
    REFCNT = 1
    FLAGS = (PADBUSY,PADMY,OBJECT,POK,pPOK)
    IV = 0
    NV = 0
    PV = 0x80f2bb0 "foo"\0
    CUR = 3
    LEN = 4
    STASH = 0x81039b8       "Tie::Foo"
```

```
MG_VIRTUAL = &PL_vtbl_packelem
MG_TYPE = 'q'
MG_FLAGS = 0x02
  REFCOUNTED
MG_OBJ = 0x80e58c8
SV = RV(0x80fb6f8) at 0x80e58c8
  REFCNT = 1
  FLAGS = (ROK)
  RV = 0x8103a00
  SV = PVMG(0x8102450) at 0x8103a00
    REFCNT = 1
    FLAGS = (PADBUSY,PADMY,OBJECT,POK,pPOK)
    IV = 0
    NV = 0
    PV = 0x80f2bb0 "fop"\0
    CUR = 3
    LEN = 4
    STASH = 0x81039b8       "Tie::Foo"
```

4.3. Array Variables

A Perl array is an array of scalar variables and at the C level an Array Value (AV) is fundamentally an array of SV's. An AV is implemented by using a structure that is the same as that used for a SvPVMG (defined in `av.h` as type xpvav) except that three additional fields are present:

ALLOC

: points to an array of SV's (in fact since this is C it points to an array of pointers to SV structures [an SV**]),

ARYLEN

: points to a magic SV that is responsible for dealing with the `$#array` Perl construct

Flag

: an extra array flag variable controlling whether the elements should have their reference counter decremented when the variable is removed from the array (this is normally true but the `@_` array is an example where this does not happen for example).

Also, the first three fields of the structure now have different names with `PVX` becoming `ARRAY`, `CUR` becoming `FILL` and `LEN` becoming `MAX`.

```
ANY => ARRAY          position of first element
       FILL           number of elements in AV - 1
       MAX            total number of elements
       IVX => not used
       NVX => not used
       MAGIC
```

```
        STASH
        ALLOC =>  Array of SV*  [0,1,2,3,4,5,6,7,8]
        ARYLEN => Magic SV
        FLAGS
```

Note that the `ARRAY` field points to the first valid element of the Perl array but the `ALLOC` field points to the first element of the C array. Usually `ARRAY` and `ALLOC` point to the same thing but similar to the OOK trick described earlier (Section 4.1.8)the `ARRAY` field can be used to efficiently shift elements off the array without adjusting memory requirements simply by incrementing the pointer. Similarly, elements can be popped off the top of the array by decrementing `FILL`.

Example 4-2. `Devel::Peek` of `@a`

```
use Devel::Peek;
@a = qw(a b);                          ❶
Dump(\@a);
```

❶ 2 element array

```
SV = RV(0x80fb6f8) at 0x80e5910
  REFCNT = 1
  FLAGS = (TEMP,ROK)                   ❶
  RV = 0x80f9c90
  SV = PVAV(0x80fcba0) at 0x80f9c90
    REFCNT = 2                         ❷
    FLAGS = ()
    IV = 0
    NV = 0
    ARRAY = 0x80ec048
    FILL = 1                           ❸
    MAX = 3                            ❹
    ARYLEN = 0x0                       ❺
    FLAGS = (REAL)
    Elt No. 0
    SV = PV(0x80e5b04) at 0x80e5838    ❻
      REFCNT = 1
      FLAGS = (POK,pPOK)
      PV = 0x80faaa8 "a"\0
      CUR = 1
      LEN = 2
    Elt No. 1
    SV = PV(0x80e5b28) at 0x80e58c8    ❼
      REFCNT = 1
      FLAGS = (POK,pPOK)
      PV = 0x80f4820 "b"\0
      CUR = 1
      LEN = 2
```

❶ These flags indicate that the SV passed to the `Dump` function is a reference to another SV and that it is temporary.

❷ The reference count is 2 because there is one reference from `@a` itself and one from the reference passed to `Dump`.

❸ Index of highest entry.

❹ Highest index that can be stored without allocating more memory.

❺ This pointer is null until $# is used for this array.

❻ This is the first element: an SvPV containing the letter "a".

❼ This is the second element: an SvPV containing the letter "b".

4.4. Hashes

Hashes (HV's) are the most complex data structure inside Perl and are used by many parts of the internals. There is no equivalent to an associative array in C so a Perl hash is implemented as an array. A simple associative array could be arranged as an array where alternating elements are the key and the value and indeed this is one way of populating a hash in Perl:

```
%hash = ( 'key1', 'value1', 'key2', 'value2');
```

The problem with this approach is that value retrieval is very inefficient for large hashes since the entire array must be searched in order to find the correct key:

```
for ($i=0; $i<= $#arr; $i+=2) {
  return $arr[$i+1] if $arr[$i] eq $key;
}
```

A more efficient approach is to translate the key into an array index which would result in a hash lookup that is almost as fast as an array lookup. The number generated from the key is called a *hash* and this gives the data structure its name. In Perl 5.6 a hash number is generated for each key using the following algorithm (the specific algorithm is modified slightly as perl versions change):

```
use integer;
$hash = 0;
foreach my $i ( 0..(length($key)-1)) {
    $hash = $hash * 33 + ord( substr($key,$i,1) );
}
$hash += ( $hash >> 5 );
```

In general this hash number is very large and is translated into an index by calculating the bitwise AND of this hash number and the index of the largest entry of the array (the size of the array minus one since the index starts counting at zero):

```
$index = $hash & $#array;
```

Since the array sizes used for hashes are always chosen to be a number 2^N-1 (a binary number containing all 1's) this guarantees that the index will be between 0 and the maximum allowed value. Unfortunately,

this technique does *not* guarantee that the index will be unique for a given hash number as only some of the bits from the hash number are used in the comparison. This problem is overcome by chaining together hash entries with the same index such that each hash entry with the same index has a data field containing a reference to the next field in the list. When searching for a specific key the algorithm is modified slightly so that it first determines the index from the hash and then goes through the chain (known as a *linked list*) until the correct key is found; still much faster than searching every entry since the search is reduced simply to those entries with a shared index. This technique is called *collisional hashing* since a single index is not unique.

```
[diagram of the array with hash entries]

  KEYS => Number of keys in hash (4 in this case)
  FILL => Number of slots occupied in array (2 in this case)
  MAX  => Size of array (7 in this case)
  ARRAY =>   0  1  2  3  4  5  6  7
                |        |
                HE1      HE2
                           |
                         HE3
                           |
                         HE4
```

A hash entry (HE) has the following structure:

```
$HE = {
       NEXT => $HE_next,          ❶
       HEK  => {                  ❷
                HASH => $hash,
                LEN  => length($key),
                KEY  => $key,
               }
       VAL  => $SV,               ❸
      };
```

❶ Pointer to the next hash entry in the list

❷ Hash Entry Key structure. This is purely a function of the hash key and therefore does not change across hashes.

❸ SV containing the actual value of the hash entry

In a perfect hash the number of hash entries matches the size of the array and none of them point to other hash entries. If the number of elements in the list exceeds the size of the array the assumption is that too many of the elements are in linked lists, where key searches will take too long, rather than evenly spread throughout the array. When this occurs the size of the array is doubled and the index of every hash entry recalculated. This does not involve recalculating the hash number for each element (since that simply involves the key itself) only recalculating the index which is the bitwise AND of the hash number and the new array size. If you are going to be inserting large numbers of keys into a hash it is usually more efficient to pre-allocate the array as this prevents many reorganizations as the array increases in size. From Perl this can be done using the `keys` function:

```
keys %h = 1000;
```

In the following dump, the keys have different hash numbers but translate to the same index (when the max size of the array is 7).

```
% perl -MDevel::Peek -e 'Dump({AF=>"hello",a=>52})'
SV = RV(0x80fbab8) at 0x80e5748
  REFCNT = 1
  FLAGS = (TEMP,ROK)
  RV = 0x80e5838
  SV = PVHV(0x80fd540) at 0x80e5838
    REFCNT = 2
    FLAGS = (SHAREKEYS)                          ❶
    IV = 2
    NV = 0
    ARRAY = 0x80eba08  (0:7, 2:1)                ❷
    hash quality = 75.0%                         ❸
    KEYS = 2                                     ❹
    FILL = 1                                     ❺
    MAX = 7                                      ❻
    RITER = -1
    EITER = 0x0
    Elt "a" HASH = 0x64                          ❼
    SV = IV(0x80f0b98) at 0x80e594c
      REFCNT = 1
      FLAGS = (IOK,pIOK,IsUV)
      UV = 52
    Elt "AF" HASH = 0x8ec                        ❽
    SV = PV(0x80e5b04) at 0x80e5904
      REFCNT = 1
      FLAGS = (POK,pPOK)
      PV = 0x80faa98 "hello"\0
      CUR = 5
      LEN = 6
```

❶ This flag indicates that the keys (the HEK structures) should be stored in a location visible to all hashes (`PL_strtab`)

❷ The bracketed numbers indicate how the hash is populated. In this example there are 7 slots in the array with 0 hash entries and one slot with 2 entries.

❸ Measure of the hash efficiciency. This number attempts to reflect how well filled the hash is. If many hash entries share the same slot this number is less than 100%.

❹ Total number of keys stored in the hash

❺ Number of slots filled in the array.

❻ Current size of the array

❼ Description of SV stored using key "a". The hash value is `0x64` in hex which translates to slot 4 when the hash has a size of 7.

❽ Description of SV stored using key "AF". The hash value is `0x8ec` in hex which translates to slot 4 when the hash has a size of 7.

The `RITER` and `EITER` fields are used to keep track of position when looping through the hash (for example with the `each` or `keys` functions). `EITER` contains a pointer to the current hash entry and `RITER` contains the array index of that entry. The next hash entry is then determined by first looking in the HE for the next entry in the linked list and if that is not defined incrementing `RITER` and looking into the array until another hash entry is found.

Since a particular hash key *always* translates to the same hash number, if the `SHAREKEYS` flag is set Perl uses an internal table (called `PL_strtab`) to store every hash key (HEK structure) currently in use. The key is only removed from the table when the last reference to it is removed. This feature is especially useful when using object-oriented programming since each instance of a hash object will only use a single set of keys.

4.5. Globs

In Perl a *glob* provides a way of accessing package variables that share the same name. Whereas globs were important in earlier versions of perl the design of Perl5 has meant that they are hardly ever required by Perl programmers doing *normal* tasks. In fact, as of version 5.6.0 of perl it is no longer even required to know that filehandles are globs[4] Globs are required in order to understand the internal representation of variables and this section describes how they are implemented.

A glob variable is based on the structure of magic variable (see Section 4.2) with the addition of fields for storing the name of the glob (`NAME` and `NAMELEN`), the namespace of the glob (`GvSTASH`) and the shared glob information (`GP`):

```
% perl -MDevel::Peek -e 'Dump(*a)'
SV = PVGV(0x8111678) at 0x810cd34   ❶
  REFCNT = 2
  FLAGS = (GMG,SMG)                 ❷
  IV = 0
  NV = 0
  MAGIC = 0x81116e8
    MG_VIRTUAL = &PL_vtbl_glob      ❸
    MG_TYPE = '*'                   ❹
    MG_OBJ = 0x810cd34
  NAME = "a"                        ❺
  NAMELEN = 1
  GvSTASH = 0x81004f4 "main"      ❻
  GP = 0x81116b0                    ❼
    SV = 0x810cce0                  ❽
    REFCNT = 1                      ❾
    IO = 0x0                        (10)
    FORM = 0x0                      (11)
    AV = 0x0                        (12)
```

```
HV = 0x0                        (13)
CV = 0x0                        (14)
CVGEN = 0x0
GPFLAGS = 0x0
LINE = 1                        (15)
FILE = "-e"                     (16)
FLAGS = 0x0
EGV = 0x810cd34 "a"         (17)
```

❶ A glob structure is called a PVGV.

❷ These flags indicate that set and get magic are available.

❸ This is the name of the table containing the glob `get`/`set` functions.

❹ The type of magic is "*" indicating that this is a glob.

❺ This is the name of the glob. All package variables of this name will be contained in this glob.

❻ This is the namespace to which this glob belongs. Each namespace has its own set of globs. Package variables are implemented using globs and stashes. Stashes are discussed in Section 4.6.

❼ The GP structure can be shared between one or more GV's. This structure contains the variable information specific to a particular glob. They are separate so that multiple GV's can share a single GP in order to implement variable aliasing.

❽ This contains a reference to the SV stored in `$a`.

❾ The reference count of the GP structure. This increases each time a variable is aliased.

(10) This contains a reference to the filehandle stored in `a`. This is currently set to `0x0` indicating that there is no filehandle of this name.

(11) This contains a reference to the format stored in `a`. This is currently set to `0x0` indicating that there is no format of this name.

(12) This contains a reference to the array stored in `@a`. This is currently set to `0x0` indicating that there is no array of this name.

(13) This contains a reference to the hash stored in `%a`. This is currently set to `0x0` indicating that there is no hash of this name.

(14) This contains a reference to the subroutine stored in `&a`. This is currently set to `0x0` indicating that there is no subroutine of this name.

(15) This is the line number of the file where the glob first occurred.

(16) This is the name of the file that declared the glob.

(17) This is the name and reference of the glob that created this GP structure. This is used to keep track of the original variable name after aliasing.

Example 4-3 further illustrates how globs work by showing how aliasing affects the glob structure:

Example 4-3. Glob aliasing

```
use Devel::Peek;
open(A,"/dev/null");                                    ❶
$A = 5;                                                 ❶
@A = (2,3,4);                                           ❶
sub A { 1; }                                            ❶
*b = *A;                                                ❷
Dump(*b);
```

❶ Use the variable named `A` to store different perl types.

❷ Make `b` an alias of `A`.

```
SV = PVGV(0x81082b0) at 0x811a0d8
  REFCNT = 3                                            ❶
  FLAGS = (GMG,SMG,MULTI)                               ❷
  IV = 0
  NV = 0
  MAGIC = 0x8108718
    MG_VIRTUAL = &PL_vtbl_glob
    MG_TYPE = '*'
    MG_OBJ = 0x811a0d8
    MG_LEN = 1
    MG_PTR = 0x8108738 "b"                              ❸
  NAME = "b"                                            ❹
  NAMELEN = 1
  GvSTASH = 0x80f8608 "main"
  GP = 0x810a350
    SV = 0x81099d0
    REFCNT = 2                                          ❺
    IO = 0x8104664                                      ❻
    FORM = 0x0
    AV = 0x8109a30                                      ❻
    HV = 0x0
    CV = 0x811a0f0                                      ❻
    CVGEN = 0x0
    GPFLAGS = 0x0
    LINE = 2
    FILE = "glob.pl"
    FLAGS = 0x2
    EGV = 0x8104598 "A"                                 ❼
```

❶ This is the reference count of the GV `*b` and is distinct from GV `*A`. This has 3 references to it: one in the main program, one from the argument passed to `Dump>` and ...

* *there always seems to be at least 2 so what is the first reference? Presumably it is from the symbol table itself although surely *A covers that.*

❷ In addition to the flags decribed previously there is an additional flag to indicate that more than one variable is stored in the glob.

❸ This is the name of the glob associated with the magic structure.

❹ This is the name of the GV that is being listed.

❺ Reference count of this GP structure. In this case it is being used by `*A` and `*b` so the reference count is 2.

❻ Pointers to the variables that were created using this package name. There is a scalar, a filehandle, an array and a subroutine all sharing a name.

❼ The name of the first GV to use this GP. In this case `A`.

4.6. Namespaces - Stashes

Namespaces are used by Perl to separate global variables into groups. Each global variable used by Perl belongs to a namespace. The default namespace is `main::`[5] (compare this with the `main` in C) and all globals belong to this namespace unless the `package` keyword has been used or unless a namespace is explcitly specified:

```
$bar = "hello";                                    ❶
$Foo::bar = 3;                                     ❷

package Foo;
$bar = 2;                                          ❷
```

❶ Sets `$main::bar`

❷ Sets `$Foo::bar`

Internally, namespaces are stored in *symbol table hashes* that are called *stashes*. They are implemented exactly as the name suggests. Each namespace (or symbol table) is made up of a hash where the keys of the hash are the variable names present in the namespace and the values are GVs containing all the variables that share that name. Stashes can also contain references to stashes in related namespaces by using the hash entry in the GV to point to another stash; Perl assumes that any stash key ending in `::` indicates a reference to another stash. The root symbol table is called `defstash` and contains `main::`. In order to look up variable `$a` in package Foo::Bar the following occurs:

1. Starting from `defstash` look up `Foo::` in the `main::` stash. This points to the `Foo::` glob.
2. Now look up the HV entry in the glob. This points to the `Foo::` stash.

3. Look up `Bar::` in this stash to retrieve the GV containing the hash for this stash.
4. Look up `a` in the `Bar::` hash to get the glob.
5. Dereference the required SV.

As you can see, finding package variables in nested namespaces has a large overhead!

* *There is an excellent diagram of this in illguts*

One final thing to note about hashes is that Perl does not hide the implementation details. The symbol table hashes can be listed just like any other Perl hash:

```
% perl -e '$Foo::bar = 2; print $Foo::{bar},"\n"'
*Foo::bar
```

Here the `bar` entry in the hash `%Foo::` is a glob named `Foo::bar` that contains the scalar variable that has a value of `2`. This can be proven by aliasing this hash to another glob:

```
% perl -e '$Foo::bar = 2; *x = $Foo::{bar}; print "$x\n"'
2
```

* *Do we mention how locals work? This probably comes under discussion of stacks and ops*

4.7. Lexical "my" variables

Lexical variables are associated with code blocks rather than namespaces so they are implemented in a very different way to globals. Each code block (a CV, see Section 4.8) contains a reference to an array (an AV) of scratch pads. This array is called a *padlist*. The first entry in this padlist is a scratch pad (also an AV) that lists all the lexical variables used by the block (not just those that were declared in the block). The names contain the full variable type (i.e. $, %, @) so that `$a` and `@a` have different entries. The second scratch pad is an array that contains the values for each of these variables. The main complication occurs with recursion. In order that each recursive call does not trash the previous contents of the variables in the scratch pads for that block a new entry in the padlist is created for each level of recursion.

```
CV ----> padlist ---->   [  $a , %a,  $z,  @z ... ]  Lex var names
                  |
                  |-->   [   5 ...                ] scratch pad 1
                  |
                  |-->   [                         ] scratch pad 2
```

* *This should be a diagram*

This layout means that at least 3 AV's are required to implement lexical variables in each code block. Lexical variables are faster than local variables because the scratch pads are created at compile time (since Perl knows which variables are to be associated with each block at that point) and can be accessed directly.

* *Simon: Robin from London.PM said this, which may be helpful: It still took me most of a day to *actually* figure scratchpads out, because APP neglected to mention that the scope of a lexical was indicated by cop_seq numbers stored in the NV and IV slots of the name SV.*
* *Tim: But do we want to talk about cop_seq in here since this is just explaining organization? Usage comes later*

4.8. Code blocks

Subroutines or code blocks in perl are thought of in the same way as other Perl variables. A Code Value (CV) contains information on Perl subroutines and the layout is similar to the structure of other variables with additional fields dealing with issues such as namespace (Section 4.5), pad lists (Section 4.7) and opcodes (Section 11.5):

```
% perl -MDevel::Peek -e 'sub foo {}; Dump(\&foo)'
SV = RV(0x8111408) at 0x80f86e0
  REFCNT = 1
  FLAGS = (TEMP,ROK)
  RV = 0x8109b7c
  SV = PVCV(0x811b524) at 0x8109b7c                    ❶
    REFCNT = 2
    FLAGS = ()
    IV = 0
    NV = 0
    COMP_STASH = 0x80f8608 "main"                      ❷
    START = 0x8108aa0 ===> 655                         ❸
    ROOT = 0x810b020                                   ❸
    XSUB = 0x0                                         ❹
    XSUBANY = 0                                        ❹
    GVGV::GV = 0x8109ae0 "main" :: "foo"               ❺
    FILE = "-e"                                        ❻
    DEPTH = 0                                          ❼
    FLAGS = 0x0                                        ❽
    PADLIST = 0x8109b28                                ❾
    OUTSIDE = 0x80f8818 (MAIN)
```

❶ The structure is of type PVCV

❷ This is the location and name of the symbol table hash that is in effect when this subroutine executes. In this example there are no packages declared and the `main::` stash is in scope.

❸ These items refer to the actual opcodes that are used to implement the subroutine. These are explained in Section 11.5

❹ When this subroutine represents an external C function these fields contains a pointer to that function and other related information. See Chapter 2 for more information on this topic

❺ This is a pointer to the GV that contains this subroutine. Recall that a glob can contain references to all the Perl variable types, in this case the GV is named `*main::foo`.

❻ The name of the file that defined the subroutine. This subroutine was created on the command line.

❼ This contains the recursion depth of the subroutine. For each level of recursion this number increments and this allows the correct entries in the padlist to be retrieved.

❽ This is the actual value of the flags field that is expanded in words at the top of the dump

❾ This is a reference to the padlist containing all the lexical variables required by this subroutine.

The following flags are of particular interest when examining CVs:

ANON

Indicates that the subroutine is anonymous (e.g. **`$x = sub { 1; };`**). When this flag is set the GV associated with the CV is meaningless since the subroutine is not present in any stash.

LVALUE

Indicates that the subroutine can be used as an lvalue.

```
use Devel::Peek;
$x = 1;
sub foo :lvalue {
  $x;
}
print foo,"\n";
foo = 5;
print foo,"\n";

Dump(\&foo);
```

would show:

```
1                                                          ❶
5                                                          ❷
SV = RV(0x8109998) at 0x8100dec
  REFCNT = 1
  FLAGS = (TEMP,ROK)
  RV = 0x810d4f8
  SV = PVCV(0x8109044) at 0x810d4f8
    REFCNT = 2
    FLAGS = (LVALUE)                                       ❸
    IV = 0
    NV = 0
    COMP_STASH = 0x8100d14 "main"
    START = 0x8113490 ===> 1178
    ROOT = 0x8130d48
    XSUB = 0x0
    XSUBANY = 0
    GVGV::GV = 0x810d4d4 "main" :: "foo"
    FILE = "lvexample.pl"                                  ❹
    DEPTH = 0
    FLAGS = 0x100
```

```
PADLIST = 0x810d528
OUTSIDE = 0x8109e34 (MAIN)
```

❶ This shows the initial state of `$x`.

❷ `$x` now has a value of 5.

❸ The `LVALUE` flag is now set confirming that the subroutine can be used as an lvalue.

❹ The example was stored in a file so now the `FILE` file field contains a proper filename.

* *Not sure there is any value in discussing esoteric flags such as CLONE, CLONED, UNIQUE, METHOD, LOCKED and CONST*

4.9. Further Reading

More detailed information on the structure of perl's internal data structures can be found at the following locations:

perlguts
perlapi

The main source of Perl internals documentation are the `perlguts` and `perlapi` man pages that comes with the Perl distribution

illguts

Perl Guts Illustrated (http://gisle.aas.no/perl/illguts/) by Gisle Aas provides an alternative illustrated explanation of the internal implementation of perl variable types.

`sv.h`
`av.h`
`hv.h`
`mg.h`
`cv.h`

This is where all the structures and flags are defined. If you really want to know the details of what is happening read these C include files that come with the Perl source.

Notes

1. inside Perl the SV type is a typedef (or alias) for a structure of type sv (a struct sv) defined in the `sv.h` include file.
2. NULL is the C equivalent of `undef`
3. the actual name of the SvPV struct in the Perl include files is xpv (similarly an SvPVIV uses a struct named xpviv). For the rest of this chapter the struct name will not be explcitly stated.

4. In perl5.6.0 it is now possible to say **`open my $fh, $file`** and treat `$fh` as a normal scalar variable.
5. In all cases the `main::` namespace can be abbreviated simply to `::`

Chapter 5. The Perl 5 API

The Perl 5 API is the interface by which your C code is allowed to talk to Perl; it provides functions for manipulating variables, for executing Perl code and regular expressions, for file handling, memory management and so on.

The API is also used inside the Perl core as a set of utility functions to manipulate the environment which the Perl interpreter provides. It's also the building blocks out of which you will create extensions and by which you will drive an embedded Perl interpreter from C - hence, the functions and macros you learn about here will form the basis of the rest of the book.

This chapter is a reference to the Perl 5 API, so the example code for each function will probably contain uses of other functions not yet explained. You are encouraged to jump around the chapter, following references to later functions and jumping back to see how they're used in real code.

As much as possible, we've tried to make the example code real, rather than contrived, taking the code from XS modules and, where possible, from the Perl sources or extension modules inside the Perl core.

5.1. Sample Entry

Here is the format we will use when introducing functions:

something *(the name of the function)*.

```
char * something(int parameter);
```

A brief explanation of what the function does.

Example 5-1. Using `something`

```
if (testing)
    result = something(whatsit);                    ❶
```

❶ An explanation of how the function is used here.

5.2. SV Functions

Before we look at the functions for manipulating SVs, there are certain special values that Perl defines for us.

5.2.1. Special SVs

5.2.1.1. `PL_sv_undef`

This is the undefined value; you should use it when, for instance, you want to return `undef` rather than an empty list or a false value. Note that `PL_sv_undef` is actually the SV structure itself - since you'll usually want a pointer to an SV, you should use this as `&PL_sv_undef` whenever anything needs a SV*.

Example 5-2. `$/ = undef` in C

This example is taken from the Apache module `mod_perl`, which embeds a Perl interpreter inside the Apache web browser. We're going to look at an entire function which takes a request for a file, opens the file, reads it all into an SV, and returns a reference to it. In Perl, we'd write it like this:

```
sub mod_perl_slurp_filename {
    my ($request) = @_;
    my $sv;

    local $/ = undef;
    open FP, $request->{name};
    $sv = <FP>;
    close FP;

    return \$sv;
}
```

Now let's see how we'd write it in C.

> **Tip:** Since this is our first example, we haven't introduced any of the other functions in the Perl API yet. Don't worry; we'll provide a commentary on exactly what's going on, and you're encouraged to jump around to the entries for other functions so you know what they do. We'll also skim over some things which will be covered in further chapters which are peripheral to the points we're making.

```
SV *mod_perl_slurp_filename(request_rec *r)
{
    dTHR;
    PerlIO *fp;
    SV *insv;
```

```
    ENTER;
    save_item(PL_rs);                                    ❶
    sv_setsv(PL_rs, &PL_sv_undef);                       ❷

    fp = PerlIO_open(r->filename, "r");                  ❸
    insv = newSV(r->finfo.st_size);                      ❹
    sv_gets(insv, fp, 0); /*slurp*/                      ❺
    PerlIO_close(fp);                                    ❻
    LEAVE;
    return newRV_noinc(insv);                            ❼
}
```

❶ `PL_rs` is the record separator, and `save_item` is the equivalent to `local`; it'll save a copy of the variable away, and restore it at the end of the current 'XS block' - the code delimited by `ENTER` and `LEAVE`.

❷ We set this SV equal to `undef` using the function `sv_setsv`. (See

❸ `PerlIO_open` opens a file - the first parameter is the filename, which we extract from the request structure `r`, and the second parameter is the mode: we're opening the file for reading. This will give us a filehandle back, just like the `open` function in Perl.

❹ We now use `newSV` (see Section 5.2.2.2) to create a new SV to receive the file contents. We pass the size of the file as a parameter to `newSV` so that Perl can grow the string to the appropriate length in advance.

❺ `sv_gets` is equivalent to the readline (<>) operator in Perl - it reads a line from the specified filehandle into the SV. Since we've set the record separator to `undef`, this has the effect of slurping the entire file.

❻ Now we close the filehandle with `PerlIO_close`.

❼ Finally, we use `newRV_noinc` (see to create a reference to the SV, without increasing its reference count, and return the new reference.

See also: Section 5.2.1.2.

5.2.1.2. `PL_sv_yes` and `PL_sv_no`

These SVs represent true and false values; you can use them as boolean return values from your functions. Just like `PL_sv_undef` above, these are the actual SV structures themselves, rather than

pointers to them, so use `&PL_sv_yes` and `&PL_sv_no` in your code.

Example 5-3. Is it a number?

There's a function in `sv.c` in the Perl core called `looks_like_number` which takes an SV and decides whether or not it's a valid numeric value. In the `DBI` module, there's a wrapper around the function allowing it to take a list of items and return a list of true or false values. Here's the meat of that function:

```
for(i=0; i < items ; ++i) {
    SV *sv = ST(i);                                    ❶
    if (!SvOK(sv) || (SvPOK(sv) && SvCUR(sv)==0))      ❷
        PUSHs(&PL_sv_undef);
    else if ( looks_like_number(sv) )                  ❸
        PUSHs(&PL_sv_yes);
    else
        PUSHs(&PL_sv_no);
}
```

❶ We loop over the arguments to the function - `ST` is a special macro which gets the numbered SV from the list of arguments that Perl keeps on the argument stack. Having taken all the arguments off the stack, we'll fill it back up with the return values.

❷ If the SV isn't available, or is a string and has no length, we put `undef` onto the stack.

❸ Now we call the function in `sv.c`, and if that returns true, we put the true value, `&PL_sv_yes` onto the stack. If it's false, we put `&PL_sv_no` on instead.

See also:

5.2.2. Creation and Destruction

5.2.2.1. `get_sv`

```
SV* get_sv(char* name, bool create);
```

This function returns you the SV which represents the Perl-side variable *`name`*; if this SV doesn't currently exist in the Perl symbol table, it will be created if *`create`* is specified - otherwise you get a null pointer back. This function is the usual way to communicate with options provided by the user as

global or package Perl variables. This should be called as merely `get_sv`, but you may also see it as `perl_get_sv` in older code.

Example 5-4. Debugging XS Modules

The most common use of `get_sv` is to read parameters set by the user in Perl space. For instance, `$YourModule::DEBUG` could be set to true to turn on debugging information. This is exactly what the `Storable` module does. Let's have a look at how it sets up its debugging mode:

```
#ifdef DEBUGME                                              ❶
#define TRACEME(x)  do {                                     \
    if (SvTRUE(perl_get_sv("Storable::DEBUGME", TRUE))) \   ❷
        { PerlIO_stdoutf x; PerlIO_stdoutf("\n"); }          \❸
} while (0)                                                 ❹
#else
#define TRACEME(x)
#endif

...

TRACEME(("small integer stored as %d", siv));
```

❶ Firstly, the module has to be complied with **`-DDEBUGME`** before any debugging can take place: if it is not, `TRACEME()` simply expands to nothing - that is, it is optimized away.

❷ We now get a "handle" on the Perl-side variable `$Storable::DEBUGME`, creating it if it doesn't already exist, and then calls `SvTRUE` (See) to test whether or not it is set to a true value.

❸ If it is, the debugging message is formatted and printed to standard output using `PerlIO_stdoutf` (See), followed by a new line.

❹ The **`do { ... } while (0)`** is a preprocessor trick to make sure the code consists of one statement, so that it can be used like this:

```
if (broken)
    TRACEME(("Something is wrong!"));
```

Without the do-while trick, this would require braces as it would be more than one statement.

See also: Section 5.3.1.1, Section 5.4.1.1.

5.2.2.2. newSV and NEWSV

```
SV* newSV(STRLEN length);


SV* NEWSV(IV id, STRLEN length);
```

There are many ways to create a new SV in Perl - the most basic are `newSV` and `NEWSV`. Both take a parameter *`length`* - if this is more than zero, the SV will be a SVPV and the PV will be pre-extended to the given length. If it is zero, you'll get a SVNULL back. The `NEWSV` macro also takes an *`id`* parameter; this was intended to be used to track where SVs were allocated from in case of memory leaks, but never really got used - you can use a unique identifier there if you think it's going to get used again, or `0` if you aren't bothered.

Example 5-5. Saving `eval`'ed code

When you execute **`eval $string`**, Perl has to store the code you're evaluating away. In fact, Perl splits it up into separate lines, so that it can return error messages like this:

```
syntax error at (eval 1) line 3, at EOF
```

As you can see, Perl has split up the string and returns an error message based on the line number. It does this by putting the code into an array, using the internal `S_save_lines` function, in `pp_ctl.c` of the Perl sources:

```
S_save_lines(pTHX_ AV *array, SV *sv)
{
    register char *s = SvPVX(sv);                          ❶
    register char *send = SvPVX(sv) + SvCUR(sv);
    register char *t;
    register I32 line = 1;

    while (s && s < send) {
        SV *tmpstr = NEWSV(85,0);                          ❷

        sv_upgrade(tmpstr, SVt_PVMG);                      ❸
        t = strchr(s, '\n');                               ❹
        if (t)
            t++;
        else
           t = send;

        sv_setpvn(tmpstr, s, t - s);                       ❺
        av_store(array, line++, tmpstr);                   ❻
        s = t;
    }
}
```

❶ The string `s` contains the text in the SV - that is, the text of the Perl code we're splitting up; `send` represents the end of the string.

❷ For every row, before we reach the end of the string, we create a new SV using `NEWSV`; we give it the leak ID 85 and do not give a length, because we don't know how long or how short our line is likely to be.

❸ We upgrade the new SV to a PVMG, so that the Perl debugger can use the magic structure.

❹ The next few lines find the next newline character, and set `t` to the start of the next line.

❺ We now have a string running from `s` to `t-1` which is our line; we use `sv_setpvn` (See) to store that string into our temporary SV.

❻ Finally, we store the SV as the next element in the array we're building up, and set our string point to `t`, the start of the next line.

See also: Section 5.3.1.2, Section 5.4.1.2, Section 5.2.2.1

5.2.3. Accessing Data

* *As well as SvIV, Also need sv_setiv et al in here - TJ*

5.2.4. String Functions

5.3. AV Functions

As we know, AVs are Perl arrays; this section contains the functions which we can perform on those arrays.

5.3.1. Creation and Destruction

5.3.1.1. `get_av`

```
AV* get_av(char* name, bool create);
```

Returns the AV variable from the Perl-side array `name`: for instance, **`get_av("main::lines", FALSE)`** will return a pointer to the AV which represents `@main::lines` in "Perl space". If the boolean *`create`* is true, a new AV will be created if the Perl-side array hasn't been created yet; if it is false, the function will return a null pointer if the Perl-side array hasn't been created.

Example 5-6. Popping a Perl array from C

In the core `DynaLoader` module, the XS function `dl_unload_all_files` wants to call a Perl subroutine `DynaLoader::dl_unload_file` on all the elements of the `@DynaLoader::dl_librefs` array. This is how it does it: (From `$CORE/ext/DynaLoader/dlutils.c:39`)

```
if ((sub = get_cv("DynaLoader::dl_unload_file", FALSE)) != NULL) {❶
    dl_librefs = get_av("DynaLoader::dl_librefs", FALSE);❷
    while ((dl_libref = av_pop(dl_librefs)) != &PL_sv_undef) {❸
       dSP;
       ENTER;
       SAVETMPS;
       PUSHMARK(SP);
       XPUSHs(sv_2mortal(dl_libref));
       PUTBACK;
       call_sv((SV*)sub, G_DISCARD | G_NODEBUG);
       FREETMPS;
       LEAVE;
    }
 }
```

❶ First it calls `get_cv` (See) to attempt to fetch the subroutine.

❷ If that subroutine exists, we then call `get_av` to retrieve the AV for the Perl-side array `@DynaLoader::dl_librefs`

❸ Lines 41--51 are equivalent to this piece of Perl:

```
    while (defined($dl_libref = pop @dl_librefs)) {
        DynaLoader::dl_unload_file($dl_libref)
    }
```

We call `av_pop` (See Section 5.3.2.3) in a `while` loop to pop off SVs from the array, and store them in `dl_libref`. The following lines then set up a callback to call the subroutine - see .

See also: Section 5.4.1.1, Section 5.2.2.1,

5.3.1.2. newAV

```
AV* newAV(void);
```

Creates a brand new array in C space, completely unconnected to the Perl symbol table and thus inaccessible from Perl. This is usually what you'll want to use if you don't need any Perl code to see your array.

Example 5-7. Storing END blocks

Perl has several "special" named blocks of code: BEGIN, END, DESTROY and so on. The perlmod documentation tells us that:

> An END subroutine is executed as late as possible, that is, after perl has finished running the program and just before the interpreter is being exited, even if it is exiting as a result of a die function. . . . You may have multiple END blocks within a file--they will execute in reverse order of definition; that is: last in, first out (LIFO).

If we can have multiple blocks, what better to store them in than an array? In fact, this is exactly what Perl does: op.c:4769

```
else if (strEQ(s, "END") && !PL_error_count) {         ❶
   if (!PL_endav)
       PL_endav = newAV();                             ❷
   DEBUG_x( dump_sub(gv) );
   av_unshift(PL_endav, 1);                            ❸
   av_store(PL_endav, 0, (SV*)cv);
   GvCV(gv) = 0;               /* cv has been hijacked */
}
```

We're in the middle of a function called newATTRSUB which is called to set up one of the special blocks.

❶ s contains the name of the block - if this is END and we haven't had any errors compiling it, we've got a valid END block, which we're going to store in the array PL_endav.

❷ First, though, we check to see if that array exists yet - if the pointer points to zero, it's not there, so we create a new array using the newAV function.

❸ Since END blocks are processed in last in, first out order, we put each successive end block at the beginning of the array - we call av_unshift (See Section 5.3.2.3) to move all the elements up, and then av_store (See Section 5.3.2.2) to store the code for this block (the CV) at the first element of the array.

See also: Section 5.2.2.2, Section 5.4.1.2

5.3.1.3. av_make

```
AV* av_make(I32 length, SV** svp);
```

`av_make` is a little used function which turns a C array of SVs into a new AV. The parameters are the number of elements in the array, and a pointer to the C array.

Example 5-8. Copying an array

mod_perl makes an ingenious use of this function to copy an array:

```
#define av_copy_array(av) av_make(av_len(av)+1, AvARRAY(av))
```

As we saw in Section 4.3, `AvARRAY(av)` points to the first element of the C array held in `av`; `av_len` (See Section 5.3.3.3) returns the highest element in the array, so one more than that is the total number of elements. Passing these to `av_make` produces a new array with the same elements.

5.3.1.4. av_clear

```
av_clear(AV* av);
```

`av_clear` simply removes all of the elements from an array, decreasing the reference count of all of the SVs in it.

Example 5-9. Emptying an array

One of the things that the `Tk` module has to do is marshal data between Tcl and Perl. For instance, if we want to assign from Tcl to a Perl value, we have to first get the Perl value into an appropriate state. If we're doing a scalar assignment, we need to make sure our SV is a scalar. Here's how `Tk` does it:

```
static SV *
ForceScalarLvalue(SV *sv)
{
 if (SvTYPE(sv) == SVt_PVAV)                          ❶
  {
   AV *av = (AV *) sv;
   SV *nsv = newSVpv("",0);                           ❷
```

```
   av_clear(av);                                    ❸
   av_store(av,0,nsv);                              ❹
   return nsv;
  }
 else
  {
   return sv;
  }
```

❶ The SV that we're handed may be a real scalar, or it might be an array. We use `SvTYPE` (See) to determine its type. If it's just a scalar, we can simply return it. If it's an array, though, we have to deal with it.

❷ We create a new empty SV to receive the assignment.

❸ Next, we get rid of everything in the array: what we're effectively doing is **`@PerlArray = $TclValue`**, which will obviously have to get rid of what was in the array before.

❹ Then we store the recipient SV into the first element of the newly emptied array.

See also: Section 5.4.1.3

5.3.1.5. `av_undef`

`av_undef``(AV*` *`av`*`);`

Example 5-10. Removing and freeing an old array

❶

❶

❷

See also: ,

5.3.2. Manipulating Elements

5.3.2.1. av_fetch

```
SV** av_fetch(AV* array, I32 index, bool lvalue);
```

Retrieves an SV from an array; this is the C-side equivalent of

```
$array[$index]
```

The *lvalue* parameter should be set if we're about to store something in the specified index: if the array is not big enough, or there hasn't been anything stored in the array index yet, Perl will extend the array, make a new SV for us, store it in the specified index and return it for us to manipulate.

Example 5-11. Storing an array in a file

The `Storable` module implements persistency by storing and retrieving Perl data structures in disk files. Here we look at an abridged form of the function for storing arrays:

```
static int store_array(stcxt_t *cxt, AV *av)
{
    SV **sav;
    I32 len = av_len(av) + 1;                          ❶
    I32 i;
    int ret;

    PUTMARK(SX_ARRAY);                                 ❷
    WLEN(len);

    for (i = 0; i < len; i++) {                        ❸
        sav = av_fetch(av, i, 0);
        if (!sav) {
            STORE_UNDEF();
            continue;
        }
        if (ret = store(cxt, *sav))                    ❹
            return ret;
    }

    return 0;
}
```

❶ Since we're going to iterate over all the elements in the array, we need to know how many Perl thinks there are: we can't use the usual C trick of keeping on going until `av_fetch` returns null, because there could be a "hole" in the middle of the array where elements haven't been used. (see the example for Section 5.3.3.1) So we call `av_len` (See Section 5.3.3.3) to find the highest-numbered element and add one to get the number of elements.

❷ These two lines are internal `Storable` macros which write a header to the file saying that the next thing stored is an array.

❸ Now we can iterate over the elements of the array, calling `av_fetch` on each element. Note that `av_fetch` returns SV** - a pointer to an SV* - because there's a difference between storing the null SV in an array, and not storing an SV at all. If the return value of `av_fetch` is NULL, then there is no SV stored in that element, and `Storable` stores `undef`.

❹ If there is a scalar there, we call `store` to write it out to the file; note that we need to dereference the SV** when passing it to `store`, because `store` expects an SV.

See also: ,

5.3.2.2. `av_store`

```
SV** av_store(AV* array, I32 index, SV* value);
```

Stores an SV in an array; this is equivalent to the Perl code

```
$array[$index] = $value;
```

Example 5-12. Storing return values in an array

The core `Threads` module manages Perl threads; it uses an array for each thread to hold the thread's return status. In this (slightly abridged) extract from `threadstart`, we look at what happens when a thread terminates and sets up its return values. (`$CORE/ext/Thread/Thread.xs:130`)

```
av = newAV();                                          ❶
if (SvCUR(thr->errsv)) {                               ❷
    MUTEX_LOCK(&thr->mutex);
    thr->flags |= THRf_DID_DIE;
    MUTEX_UNLOCK(&thr->mutex);
    av_store(av, 0, &PL_sv_no);                        ❸
    av_store(av, 1, newSVsv(thr->errsv));
}
```

```
else {
    av_store(av, 0, &PL_sv_yes);                        ❹
    for (i = 1; i <= retval; i++, SP++)
        sv_setsv(*av_fetch(av, i, TRUE), SvREFCNT_inc(*SP));❺
}
```

❶ Firstly, we create a new array to hold the return values; since this does not need to be accessible in the Perl symbol table, we use `newAV`.

❷ If the thread exited with an error message, we need to update the thread's status flags to report that it died an unnatural death. (The `MUTEX` business on either side of the flag manipulation just makes sure that no other threads try and interfere while we're doing this.)

❸ Then we use `av_store` to store the return status in our array: the first element, element zero, will contain a false value (see Section 5.2.1.2) to denote failure. We call `newSVsv` to copy the error message into a new SV, and then use `av_store` once more to store our copy in element one of our return array.

❹ If everything succeeded, on the other hand, we store a true value (see Section 5.2.1.2) as the first element of the array.

❺ Here we see another way of storing an `SV` in an array: we use the `av_fetch` function with the *`lvalue`* parameter set to true to create the array element and return a pointer to the new SV; we then dereference that pointer and call `sv_setsv` to set it to what we want. Since we're storing the same SV in multiple places without taking a copy of it, we must increase the reference count of what we're storing.

Sometimes the core isn't the best example...

In the example, the return values from `av_fetch` and `av_store` are not checked - we merely assume that the operations will succeed. While this is usually a pretty safe assumption, it's not the best thing to do: if, for instance, the `av_fetch` failed for some reason, we'd attempt to dereference zero, which would cause the program to crash. Furthermore, you should check that `av_store` completed successfully if you are storing the same SV in multiple places - you are responsible for increasing the reference count of the SV before calling `av_store`, and if `av_store` returns zero, you are responsible for decreasing the reference count again.

See also: Section 5.3.2.1,

5.3.2.3. `av_pop` and `av_shift`

```
SV* av_pop(AV* array);
SV* av_shift(AV* array);
```

`av_pop` is just like the Perl function `pop`; it removes the highest element from an array and returns it.

Example 5-13. ???

❶

❶

❷

See also: ,

5.3.2.4. `av_push` and `av_unshift`

```
av_push(AV* array, SV* value);
av_unshift(AV* array, SV* value);
```

Example 5-14. ???

❶

❶

❷

See also: ,

5.3.2.5. av_delete

```
??? av_delete(??? ??);
```

Example 5-15. ???

❶

❶

❷

See also: ,

5.3.2.6. av_exists

```
??? av_exists(??? ??);
```

Example 5-16. ???

❶

❶

❷

See also: ,

5.3.3. Testing and Changing Size

5.3.3.1. av_extend

```
av_extend(AV* array, IV index);
```

Example 5-17. Assigning to an array slice

When you assign to an array slice in Perl, there's a possibility that you'll be assigning to elements that don't exist yet. For instance, in this code:

```
@array = (1,2,3,4);

@array[10, 30] = qw( hi there );
```

we only have 4 elements in our array, and we're about to store to elements 10 and 30. If we merely used `av_store`, Perl would have to extend the array twice: once to accomodate element 10, and once to accomodate element 30. Instead, Perl looks over the slice, finds the maximum element it's going to assign to, (in this case, 30) and calls `av_extend` to pre-extend the array. (`pp.c:3339`)

```
I32 max = -1;
for (svp = MARK + 1; svp <= SP; svp++) {              ❶
     elem = SvIVx(*svp);                              ❷
     if (elem > max)
         max = elem;
}
if (max > AvMAX(av))                                  ❸
    av_extend(av, max);
```

❶ Perl ensures that the elements of our slice (10 and 30) are placed on a stack, starting from `MARK + 1` and ending at `SP`. We look at each of these elements in turn.

❷ These elements are pointers to SV*s, and to find the numerical value, we dereference the pointer and call `SvIVx` on the SV to examine its IV. We compare each one against our stored maximum and, if necessary, update the maximum value.

❸ Now we know the highest element we'll be assigning to, we check to see whether this is higher than the current size of the array. If so, we need to call `av_extend` to increase the array's size.

5.3.3.2. av_fill

```
av_fill(AV* av, I32 fill);
```

Example 5-18. ???

❶

❶

❷

See also: ,

5.3.3.3. av_len

```
I32 av_len(AV* av);
```

Example 5-19. ???

❶

❶

❷

See also: ,

5.4. HV Functions

5.4.1. Creation and Destruction

5.4.1.1. get_hv

```
HV* get_hv(char* name, bool create);
```

Example 5-20. ???

❶

❶
❷

See also: Section 5.3.1.1,

5.4.1.2. newHV

```
HV* newHV(void);
```

Example 5-21. Creating a new hash

❶

❶
❷

See also: Section 5.3.1.2,

5.4.1.3. hv_clear

```
hv_clear(HV* hv);
```

Example 5-22. ???

❶

❶
❷

See also: Section 5.3.1.4, Section 5.4.1.4

5.4.1.4. hv_undef

```
void hv_undef(HV* hv);
```

Example 5-23. Freeing a hash

❶

❶
❷

See also: Section 5.3.1.5

5.4.2. Manipulating Elements

5.4.2.1. `hv_delete_ent`

```
SV* hv_delete_ent(HV* hash, SV* key, I32 flags, U32 hashval);
```

Example 5-24. ???

❶

❶
❷

See also: ,

5.4.2.2. `hv_exists_ent`

```
bool hv_exists_ent(HV* hash, SV* key, U32 hashval);
```

Example 5-25. ???

❶

❶

❷

See also: ,

5.4.2.3. `hv_fetch_ent`

```
??? hv_fetch_ent(??? ??);
```

Example 5-26. ???

❶

❶

❷

See also: ,

5.4.2.4. `hv_store_ent`

```
??? hv_store_ent(??? ??);
```

Example 5-27. ???

❶

❶

❷

See also: ,

5.5. Miscellaneous Functions

5.5.1. Package and Stash Manipulation

5.5.2. Memory Management

5.5.2.1. `New`

```
??? New(??? ??);
```

Example 5-28. ???

❶

❶

❷

See also: Section 5.5.2.4,

5.5.2.2. `Copy`

```
??? Copy(??? ??);
```

Example 5-29. ???

❶

❶

❷

See also: Section 5.5.2.3,

5.5.2.3. `Move`

```
??? Move(??? ??);
```

Example 5-30. ???

❶

❶

❷

See also: Section 5.5.2.2,

5.5.2.4. Renew

```
??? Renew(??? ??);
```

Example 5-31. ???

❶

❶

❷

See also: Section 5.5.2.1,

5.5.2.5. `Safefree`

```
??? Safefree(??? ??);
```

Example 5-32. ???

❶

❶
❷

See also: Section 5.5.2.1,

5.5.2.6. `Zero`

```
??? Zero(??? ??);
```

Example 5-33. ???

❶

❶
❷

See also: ,

5.5.3. File Handling

5.5.4. Unicode Data Handling

5.5.5. Everything Else

* *The ExtUtils typemap file uses INT2PTR so we have to make sure that is covered somewhere soonish. Not sure what the plan is for this - TJ*

Chapter 6. Advanced XS Programming

Gluing simple C functions to Perl is fairly straightforward and requires no special knowledge of the Perl internals. Chapter 2 covers the basics of XS and how to pass in and return values associated with simple C scalar types. Unfortunately, many functions and libraries have much more complicated signatures and require more work to implement a Perl interface.

This chapter covers a more advanced use of XS, dealing with topics such as structures, arrays, and callbacks. This chapter builds on Chapter 2 and will also refer to functions used in the Perl internals and described in Chapter 5.

6.1. Pointers and things

Now that we know about pointers and dynamic memory allocation we can start doing more interesting things with XS. To demonstrate some of these issues we can use the following function which concatenates two strings to a third and returns the total length of the final string [1].

```
#include <string.h>

STRLEN strconcat (char* str1, char* str2, char* outstr) {
   strcpy( outstr, (const char*)str1 );
   strcat( outstr, (const char*)str2 );
   return strlen( outstr );
}
```

We will now write an XS interface to mirror the C calling interface. The C signature of:

```
STRLEN strconcat(char * str1, char * str2, char * outstr);
```

will then translate to a Perl signature of:

```
$len = strconcat($str1, $str2, $outstr);
```

In this case we could write the XS interface as follows:

```
STRLEN
strconcat( str1, str2, outstr )
  char* str1
  char* str2
  char* outstr = NO_INIT
 OUTPUT:
```

```
  outstr
  RETVAL
```

Here the `NO_INIT` keyword is used to tell the compiler not to care about the input value of `outstr`. Remember that we have to tell **xsubpp** that we want to use the return value even though we have specified a return type.

Unfortunately, the above code will not work because our simple `strconcat` function assumes that the output string has enough space to hold the concatenated string. In the above XS segment `outstr` is typed as a pointer to a string but is not actually pointing to anything! We have to fix this by using a `CODE` block that allocates the memory for a string of the required size[2]

```
STRLEN
strconcat( str1, str2, outstr )
  char* str1
  char* str2
  char* outstr = NO_INIT
 PREINIT:                                                  ❶
  STRLEN length;
 CODE:
  length = strlen( str1 ) + strlen( str2 ) + 1;            ❷
  New(0, outstr, length, char );                           ❸
  RETVAL = strconcat( str1, str2, outstr );
 OUTPUT:
  outstr
  RETVAL
```

❶ PREINIT is used to declare additional variables

❷ Calculate the size of the required string. Don't forget the extra space for the the null character!

❸ We use `New` to allocate the memory rather than `malloc` since we have access to the Perl macros. See Chapter 3 for more details on this function.

Now this routine is becoming complicated! The `PREINIT` block is there to initialise additional variables that are required for the `CODE` section. `PREINIT` guarantees that the variable declaration will occur as soon as possible after entering the function as some compilers do not like declarations after code sections have started. It provides a nice way to separate variable declarations from code. Even worse, whilst our XS interface will now work there is still a problem since each time the routine is entered memory is allocated to `outstr` but it is never freed. XS provides a means to tidy up after ourselves by using the `CLEANUP` keyword. The cleanup code is guaranteed to run just before the C code returns control to Perl. Our final XS function should now work without memory leaks:

```
STRLEN
strconcat( str1, str2, outstr )
  char* str1
  char* str2
```

```
 char* outstr = NO_INIT
PREINIT:
 STRLEN length;
CODE:
 length = strlen( str1 ) + strlen( str2 ) + 1;
 New(0, outstr, length, char );
 RETVAL = strconcat( str1, str2, outstr );
OUTPUT:
 outstr
 RETVAL
CLEANUP:
 Safefree( outstr );
```

This example shows us how to deal with pointer types (in this case a simple string) and how to allocate and free memory using XS. It also demonstrates the wrong way to approach interface design - in a real application the string would be a return value (without the length) and would not be returned via the argument list.

6.2. Filehandles

Sometimes an external library needs to print to a user-supplied filehandle or, occasionally, opens a file and returns a filehandle to the user. If the library uses C input/output streams then it is easy to pass the C stream to and from perl with XS since by default XS knows how to deal with a FILE*, converting it to and from a perl filehandle. The following example could be used to provide an XS interface to one of the Gnu readline functions[3]:

```
int
rl_getc( file )
  FILE * input
```

This example shows that a FILE* can be treated like any other simple type, in this case the filehandle is an input argument:

```
$retval = rl_getc(FH);
```

but it is just as easy to import a stream into perl.

* *Care must be taken if the external library closes an imported file handle without perl knowning - on linux this causes a core dump (cf. XS::Typemap)*

If your external library requires a file descriptor (see Section 3.5) then you will have to use the `fileno` to extract the file descriptor from the file handle or stream (either by using the perl or the C `fileno` functions). Similarly, if you are importing a file descriptor into Perl you need to convert it to a filehandle either by using the `fdopen` C function (if you are comfortable with XS and C) or by importing the descriptor into perl as an integer and then using Perl's `open` command[4] to translate it.

```
$fd = some_xs_function();
```

```
open(FH, "<&=$fd");
```

From version 5.7.1 of Perl the I/O subsystem is completely self-contained and no longer relies on the underlying operating system for implementation. Perl itself uses a PerlIO* rather than a FILE* for all I/O operations (although in some cases a PerlIO* can be the same as a FILE*). If you are using I/O in your XS code but you are not using an external library then you should be using PerlIO* rather in preference to a FILE*. Of course, a PerlIO* is recognised automatically by XS.

6.3. Typemaps

When a variable is passed from Perl to C (or from C to Perl) it must be translated from a Perl scalar variable (Chapter 4) to the correct type expected by the C function. So far this translation has been implicitly assumed to be something that "just happens" but before we can move further into XS we have to explain *how* it happens.

The XS compiler (**xsubpp**) uses a lookup table, called a *typemap*, to work out what to do with each variable type it encounters. Perl comes with a `typemap` file that contains the common variable types and it is installed as part of Perl. On many Unix systems it can be found in `/usr/lib/perl5/5.6.0/ExtUtils/typemap`[5]. Here is a subset of that file:

```
# basic C types
int T_IV
unsigned int T_UV
long T_IV
unsigned long T_UV
char T_CHAR
unsigned char T_U_CHAR
char * T_PV
unsigned char * T_PV
size_t T_IV
STRLEN                  T_IV
time_t T_NV
double                  T_DOUBLE

############################################
INPUT
T_UV
$var = ($type)SvUV($arg)
T_IV
$var = ($type)SvIV($arg)
T_CHAR
$var = (char)*SvPV($arg,PL_na)
T_U_CHAR
$var = (unsigned char)SvUV($arg)
T_NV
$var = ($type)SvNV($arg)
T_DOUBLE
$var = (double)SvNV($arg)
```

```
T_PV
$var = ($type)SvPV($arg,PL_na)
############################################
OUTPUT
T_IV
sv_setiv($arg, (IV)$var);
T_UV
sv_setuv($arg, (UV)$var);
T_CHAR
sv_setpvn($arg, (char *)&$var, 1);
T_U_CHAR
sv_setuv($arg, (UV)$var);
T_NV
sv_setnv($arg, (double)$var);
T_DOUBLE
sv_setnv($arg, (double)$var);
T_PV
sv_setpv((SV*)$arg, $var);
```

The first section contains a list of all the C types of interest (there are many more in the actual file) along with a string describing the type of variable. As can be seen from the list this provides a many-to-one translation since many different C variable types can have the same fundamental representation via the use of typedefs (see Section 1.6.2). For example, both size_t and STRLEN are fundamentally integer types and can be represented by a T_IV in the typemap.

The second section is called “INPUT” and provides the code required to translate a Perl variable to the corresponding C type. The third section is called “OUTPUT” and does the reverse: providing code to translate C variables to Perl variables. The identifier matches the value defined in the first section and the functions are simply those described in . For example, the typemap entry to translate an SV to an integer (T_IV) uses `SvIV` to retrieve the integer from the SV and `sv_setiv` to set the integer part of an SV.

The `typemap` file may look strange because it includes Perl-style variables in C-type code. The variables `$arg`, `$var` and `$type` (and for more complex entries `$ntype`) have a special meaning in typemaps.

$arg

This is the name of the Perl SV in the Perl argument list.

$var

This is the name of the C variable that is either receiving the value from the SV or setting the value in the SV.

$type

This is the type of the C variable. This will be one of the types listed at the top of the typemap file.

$ntype

The type of the C variable with all asterisks replaced with the string “Ptr”. A char * would therefore set `$ntype` to “charPtr”. This variable is sometimes used for setting classnames or for referencing helper functions.

$Package

The Perl package associated with this variable. This is the same as the value assigned to the `PACKAGE` directive in the XS file.

$func_name

This is the name of the XS function.

$argoff

The position of the argument in the argument list. Starts counting at 0.

You will find that in many cases you will need to add extra typemap entries when creating XS interfaces. Rather than add to the standard typemap all that is required is to create a file called `typemap` in your module directory and add entries in the same format as that used in the default typemap above. The `makefile` that is generated from `Makefile.PL` will automatically include this typemap file in the XS processing.

6.4. The Argument Stack

In Perl arguments are passed into and out of subroutines as lists. The list is called an *argument stack*: arguments are pushed onto the stack by the caller and shifted off the stack by the subroutine. Any Perl program will demonstrate this behaviour:

```
my ($sum, $diff) = sumdiff( 5, 3 );

sub sumdiff {
  my $arg1 = shift; # implicitly shifts of @_
  my $arg2 = shift;

  return ( $arg1 + $arg2, $arg1 - $arg2 );
}
```

Perl keeps track of the number of arguments on the stack that are meant for the current subroutine (i.e. the size of `@_`)

* *Do we want to explain how or is that too detailed for this section?*

XS routines use the exact same technique when passing arguments from Perl to the XS layer. In our discussion so far this has happened automatically and the arguments from the stack have been processed using the provided typemap. Perl provides the `ST` macro to retrieve the SV on the stack. `ST(0)` is

equivalent to `$_[0]`, `ST(1)` is equivalent to `$_[1]` etc. Indeed in the typemap definitions described in the previous section `$arg` is actually replaced by `ST()` macros corresponding to the required stack position. More details on this replacement can be found in Section 6.10

So far we have just looked at XS functions that either modify input arguments (ultimately using the `ST` macros) and/or return a single value. It is also possible to write XS functions that take full control of the argument stack and this chapter contains examples on how to achieve this using PPCODE.

6.5. C Structures

C structures (Section 3.4) are used in many libraries to pass related blocks of data around. This section shows how you can handle them in XS. The choice you make depends entirely on the way in which the structure is to be used.

6.5.1. ...as black boxes

If you don't want to look inside the structure (or are not allowed to) then one approach to structures is simply to return the pointer to the structure and store it in Perl scalar. Usually, the pointer is then used as an argument for other library routines. As a simple example of this we will provide an XS interface to some of the POSIX functions that deal with time. They are:

```
struct tm * gmtime(const time_t *clock);
```

Returns a `tm` structure (using Universal Time) for a given Unix time (e.g. the output from the Perl `time` function). This is the routine used for the Perl `gmtime` builtin.

```
time_t timegm(struct tm * tm);
```

Convert a `tm` structure to a Unix time.

```
size_t strftime(char * s, size_t maxsize, char * format, struct tm * tm);
```

Convert a `tm` structure to a formatted string.

In other words, to use the above functions we don't need to know the contents of the `tm` structure. For the purposes of this example we will place the XS routines into a Perl module called `Time`. The first step is to create the module infrastructure:

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

The first function to implement is `gmtime` since that returns the base structure. Here is a first attempt at the XS code:

```
struct tm *
gmtime( clock )
  time_t &clock                                          ❶
```

❶ The ampersand here indicates that we wish to pass a pointer to the `gmtime` function. Perl first copies the argument into the variable `clock` and then passes the pointer to the function. Without the ampersand the default behaviour would be to pass the value to the function.

If we attempt to build this (after running **`perl Makefile.PL`**) we get the following error:

```
Error: 'struct tm *' not in typemap in Time.xs, line 11
Please specify prototyping behavior for Time.xs (see perlxs manual)
make: *** [Time.c] Error 1
```

The problem is that Perl does not now how to deal with a pointer to a `tm` structure since it is not present in the default typemap file. To fix this we have to create a typemap file (called `typemap`) and place it in the build directory. Since we are just interested in the pointer (and not the contents) the `typemap` just needs to contain the following:

```
struct tm *            T_PTR
```

* *perlxs indicates that a tab is required between the "*" and T_PTR but xsubpp runs fine with just spaces and, looking at the code, does not even have a \t in there*

`T_PTR` tells Perl to store the pointer address directly into a scalar variable. After saving this file the module should build successfully. We can test this with the following:

```
% perl -Mblib -MTime -e 'print Time::gmtime(time)'
1075295360
```

Your actual result will vary since this is a memory address. Now that we have a pointer we can pass it to a function. `timegm` would look like this:

```
time_t
```

```
timegm( tm )
  struct tm * tm
```

and could be used as follows:

```
use Time;
$tm = Time::gmtime( time() );
$time = Time::timegm( $tm );
```

The default implementation of `strftime` looks like this:

```
size_t
strftime( s, maxsize, format, tm )
  char * s
  size_t maxsize
  char * format
  struct tm * tm
 OUTPUT:
  s
```

Unfortunately, although this does work okay there are serious problems with the interface to `strftime` as implemented above. To use it you would have to presize the output string and provide the length of the output buffer (including the C-specific terminating null character) two things that people are used to in C but which are completely unacceptable in a perl interface:

```
use Time;
$tm = Time::gmtime( time() );
$s = "         ";  # presize output buffer
$maxsize = length($s)+1; # length must account for the null
$format = "%D";
$len = Time::strftime($s, $maxsize, $format, $tm);
print "length=$len output=$s\n";
```

A much better Perl interface would be something like:

```
$s = strftime($tm, $format);
```

where we have removed the input buffer requirement completely and rearranged the argument order to place the `tm` structure at the start of the list. One way of implementing this interface is to write a pure perl wrapper (placed in the .pm file) that deals with the presized buffer and then calls the XS code. Whilst this is sometimes easier to implement (especially if you are making use of Perl functionality), often it is more efficient to rewrite the XS layer using CODE: blocks:

```
char *
strftime( tm, format )
  struct tm * tm
  char * format
```

```
PREINIT:
 char tmpbuf[128];
 size_t len;
CODE:
 len = strftime( tmpbuf, sizeof(tmpbuf), format, tm);
 if (len > 0 && len < sizeof(tmpbuf)) {
    RETVAL = tmpbuf;
 } else {
    XSRETURN_UNDEF;
 }
OUTPUT:
 RETVAL
```

This is much better but still not perfect. The problem now is that we don't know the required size of the output buffer before calling `strftime`. In the above example we simply allocate 128 characters and hope that that is enough. For most cases it will be but if a large format is supplied this function will currently just return `undef`. One way to overcome this is to check the return value of `strftime` and increase the buffer size until it is large enough and this is exactly the way that `POSIX::strftime` is implemented in standard Perl.[6]

The example so far has demonstrated how to pass a structure pointer from and to a C library but the interface implemented above has some remaining issues. In the following sections we address some of these problems.

6.5.1.1. T_PTR versus T_PTRREF

Using a scalar to store a memory address is dangerous since it is possible that, inadvertently, the Perl program may change the value of an existing pointer (maybe by treating it as a normal number) or may pass an undefined value (0) into the `timegm` or `strftime` functions. If any of these things occur the program will crash with a memory error since the value will no longer point to a valid memory location. The best way of dealing with this problem is to use an object interface (see Section 6.5.2) but failing that another option is to return a reference to a scalar containing the pointer value rather than the scalar itself. The `T_PTRREF` typemap designation does exactly that:

```
T_PTR
        sv_setiv($arg, (IV)$var);
T_PTRREF
        sv_setref_pv($arg, Nullch, (void*)$var);
```

This has the advantage that the function will not run unless a scalar reference is passed in (much harder to do by mistake).

6.5.1.2. Default arguments

Rather than always forcing the time to be supplied a cleaner approach is to allow for the current time to be assumed if no arguments are present. This matches the behaviour of the `gmtime` builtin.[7]

```
struct tm *
gmtime( ... )                                                    ❶
 PREINIT:                                                        ❷
  time_t clock;
 CODE:
  if (items > 1)                                                 ❸
     Perl_croak(aTHX_ "Usage: Time::gmtime( [time] )");          ❹
  else if (items == 1)
     clock = (time_t)SvNV(ST(0));                                ❺
  else
     clock = time( NULL );                                       ❻

  RETVAL = gmtime( &clock );                                     ❼
 OUTPUT:
  RETVAL
```

❶ The ellipsis (`...`) indicates to the XS compiler that the number of input arguments is not known. In this example there are no required arguments whereas a signature of **`gmtime( clock, ...)`** could be used to indicate that there will be at least one argument.

❷ There are no required arguments so we have to explicitly declare the `clock` variable since **xsubpp** can no longer determine this from the argument list.

❸ The `items` variable is supplied by the XS system and contains the number of input arguments waiting on the stack. Here we are checking to see if more than one argument has been supplied.

❹ If more than one argument has been supplied we stop the program with a `Perl_croak`. This provides similar functionalty to the `croak` provided by the `Carp` module.

❺ The ellipsis implies that we have to do our own argument processing. If there is a single argument the numeric value is retrieved from the first argument on the stack (`ST(0)`). This code is identical to that found in the standard typemap file (see Section 6.3).

❻ If there are no arguments the current time is obtained using the system `time` function. The `NULL` macro is used to indicate that we are only interested in a return value.

❼ Run the `gmtime` using the value stored in `clock` and store the pointer in `RETVAL`.

6.5.1.3. Static memory

A more worrying problem is associated with the `gmtime` itself. This always uses the same structure (and therefore returns the same memory address) so each time it is called it overwrites the answer from a previous call. This is evident in the following example:

```
use Time;
$tm1 = Time::gmtime( time() );
print "First time: ",Time::timegm( $tm1 ), "\n";
$tm2 = Time::gmtime( time() + 100 );

print "First time (again): ",Time::timegm( $tm1 ), "\n";
print "Second time: ",Time::timegm( $tm2 ), "\n";
```

which prints

```
First time: 983692014
First time (again): 983692114
Second time: 983692114
```

This may cause confusion unless very carefully documented (not everyone is an expert C programmer used to these oddities). On systems where it is available one solution is to use `gmtime_r`, the reentrant (thread-safe) version of this function, since that takes the address of the structure as an argument:

```
struct tm *
gmtime( clock );
  time_t clock;
 PREINIT:
  struct tm * tmbuf;
 CODE:
  New( 0, tmbuf, 1, struct tm );
  RETVAL = gmtime_r( &clock, tmbuf );
 OUTPUT:
  RETVAL
```

A more general solution (but not thread-safe) is to copy the result from `gmtime` into a new structure each time:

```
struct tm *
gmtime_cp( clock );
  time_t clock;
 PREINIT:
  struct tm * tmbuf;
  struct tm * result;
 CODE:
  result = gmtime( &clock );
  New( 0, tmbuf, 1, struct tm );
  StructCopy( result, tmbuf, struct tm);
  RETVAL = tmbuf;
 OUTPUT:
  RETVAL
```

Both these techniques overcome the problem with `gmtime` but they both introduce a memory leak. The reason for this is that the memory allocated for the new structure (using the `New` function) is never given back to the system. The Perl scalar containing the memory address attaches no special meaning to it; if

the variable goes out of scope the SV is freed without freeing the memory. The C way to deal with this is to provide a function that can be called when the structure is no longer needed (an XS function that simply calls `Safefree`).

The more Perl-like way to handle this problem is to turn the structure into an object such that Perl automatically frees the memory when the variable goes out of scope. This approach is discussed in Section 6.5.2. Alternatively, if the structure is fairly simple (and does not contain pointers to other variables) it is possible to either copy the memory contents directly into a perl variable by using the T_OPAQUEPTR typemap entry or to copy the contents into a perl hash. These both have the advantage of allowing perl to keep track of memory management rather than the programmer. The hash approach is partially discussed in Section 6.5.3. Both approaches allow the caller to modify the contents of the hash between calls, it is no longer a black box, and the hash approach does required more work from the XS programmer.

6.5.2. ...as objects

An object can be thought of as some data associated with a set of subroutines (methods). In many libraries, C structures take on the same role as objects and Perl can treat them as such. An OO interface to the time functions described earlier may look something like (the use of `new` as a constructor is purely convention):

```
use Time;

$tm = new Time( time() );
$time = $tm->timegm;
$s = $tm->strftime( $format );
```

The changes required to the existing `Time` module to get this behaviour are not very extensive. In this section we will modify the `Time` module so that it matches the above interface.

The single most important change is to modify the typemap entry to the following to use T_PTROBJ. T_PTROBJ is similar to T_PTTREF except that the reference is blessed into a class. Here is the OUTPUT entry in the typemap file:

```
T_PTROBJ
        sv_setref_pv($arg, \"${ntype}\", (void*)$var);
```

By default the reference will be blessed into class `$ntype` which translates to `struct tmPtr`! A class containing a space is not very helpful since the XS compiler does not know how to handle them. We can get around this problem in two ways. We can either create a new OUTPUT entry (and corresponding INPUT entry) that uses a hard-wired package name:

```
struct tm *     T_TMPTROBJ

INPUT:
```

```
T_TMPTROBJ
        if (sv_derived_from($arg, \"TimePtr\")) {
            IV tmp = SvIV((SV*)SvRV($arg));
            $var = INT2PTR($type,tmp);
        }
        else
            Perl_croak(aTHX_ \"$var is not of type TimePtr\")

OUTPUT:
T_TMPTROBJ
        sv_setref_pv($arg, \"TimePtr\", (void*)$var);
```

or we can create a new variable type and associate that with T_PTROBJ. We will adopt the latter technique since it is more robust against changes to the behaviour of typemap entries. This requires the following line to be added before the MODULE line in the XS file to generate a new type Time as an alias for struct tm:

```
typedef struct tm Time;
```

and the typemap file modified to include:

```
Time *     T_PTROBJ
```

Now wherever we used struct tm we now use Time. Here is the new constructor (including all the changes suggested earlier):

```
Time *                                                    ❶
new( class, ... );                                        ❷
  char * class
 PREINIT:
  time_t clock;
  Time * tmbuf;
  Time * result;
 CODE:
  if (items > 2)
     Perl_croak(aTHX_ "Usage: new Time( [time] )");
  else if (items == 2)
     clock = (time_t)SvNV(ST(1));                         ❸
  else
     clock = time( NULL );

  result = gmtime( &clock );
  New( 0, tmbuf, 1, Time );
  StructCopy( result, tmbuf, Time);
  RETVAL = tmbuf;
 OUTPUT:
  RETVAL
```

❶ The return value is now a pointer to Time rather than a pointer to a struct tm.

❷ The function name has changed to `new` and now there is one required argument (the class name) as well as the optional second argument. This is no different to any other perl method.

❸ The Unix time is now the second argument (`ST(1)`) rather than the first.

The changes are minor compared to the previous non-OO version and caused entirely by the extra argument. [8] If you build this module we can check the result of the constructor:

```
% perl -Mblib -MTime -e 'print new Time()'
TimePtr=SCALAR(0x80f87b8)
```

As expected it returns an object blessed into class `TimePtr`. The complication now is that the timegm and strftime methods must be put into the `TimePtr` package and not the default `Time` namespace. We do this by adding an addditional MODULE directive after the constructor and then adding the methods:

```
MODULE = Time PACKAGE = TimePtr

time_t
timegm( tm )
  Time * tm

char *
strftime( tm, format )
  Time * tm
  char * format
 PREINIT:
  char tmpbuf[128];
  size_t len;
 CODE:
  len = strftime( tmpbuf, sizeof(tmpbuf), format, tm);
  if (len > 0 && len < sizeof(tmpbuf)) {
     RETVAL = tmpbuf;
  } else {
     XSRETURN_UNDEF;
  }
 OUTPUT:
  RETVAL
```

Other than the extra package declaration these definitions are *exactly* the same as those used previously. Perl automatically finds these functions and passes the structure pointer (the object) as the first argument.

There are at least two reasons to prefer the OO interface over storing the plain pointer [9]:

Type safety

: The INPUT typemap entry for T_PTROBJ includes a check for the class of the input variable. This guarantees that the object is of the correct type. Unless a programmer really tries hard this will

prevent strange values (with even stranger memory addresses) being passed to the C layer and causing segmentation faults.

Destructors

As in any Perl class when the object goes out of scope and is freed Perl will call a DESTROY. XS implementations of perl classes behave in exactly the same way. Recall that in the previous implementation the `gmtime` generated a memory leak because it was not possible to automatically free the allocated to the structure. As written the current object implementation also has the problem but it can be fixed simply by adding a DESTROY function to the `TimePtr` class:

```
void
DESTROY( tm )
  Time * tm
 CODE:
  printf("Calling TimePtr destructor\n");
  Safefree( tm );
```

Now whenever a `TimePtr` object is freed the destructor will be called and the memory will be freed.

6.5.3. ...as hashes

If the main reason for the structure is to group return values that are of interest then you should consider unpacking the structure into either a Perl hash or a list that can be converted into a hash. We will demonstrate both these techniques by extending our `Time` module so that it uses a Perl hash rather than a structure pointer. For clarity, the examples will not include support for defaulting of the time.

6.5.3.1. Returning a hash reference

One way of returning a hash is to return the reference to a hash:

```
$hash = Time::gmtime_as_href( time );
print "Day is ", $hash->{"mday"}, "\n";
```

The XS code required for this is:

```
HV *                                                    ❶
gmtime_as_href( clock )                                 ❷
  time_t clock                                          ❸
 PREINIT:                                               ❹
  HV * hash;
  struct tm * tmbuf;
 CODE:
  /* Run gmtime */
  tmbuf = gmtime( &clock );                             ❺
```

```
  hash = newHV();                                                  ❻

  /* Copy struct contents into hash */
  hv_store(hash, "sec",  3, newSViv(tmbuf->tm_sec), 0);            ❼
  hv_store(hash, "min",  3, newSViv(tmbuf->tm_min), 0);
  hv_store(hash, "hour", 4, newSViv(tmbuf->tm_hour), 0);
  hv_store(hash, "mday", 4, newSViv(tmbuf->tm_mday), 0);
  hv_store(hash, "mon",  3, newSViv(tmbuf->tm_mon), 0);
  hv_store(hash, "year", 4, newSViv(tmbuf->tm_year), 0);
  hv_store(hash, "wday", 4, newSViv(tmbuf->tm_wday), 0);
  hv_store(hash, "yday", 4, newSViv(tmbuf->tm_yday), 0);

  RETVAL = hash;                                                   ❽
 OUTPUT:
  RETVAL
```

❶ Here we set the return value of our function to be a pointer to a hash. Remember that the argument stack can only contain scalar types so the typemap will automatically convert this to a hash reference when it is placed on the stack.

❷ We call this `gmtime_as_href` to distinguish it from the normal `gmtime` function.

❸ Here we just pass in the time in seconds rather than providing a means for defaulting to the current time.

❹ This block declares the additional variables that will be required. `hash` is a pointer to an HV, `tmbuf` is declared as a pointer to the struct that will contain the result from the `gmtime` call. Since the hash will also be returned we could have used `RETVAL` throughout rather than creating an extra variable but the explicit use of a variable name is sometimes clearer.

❺ Call `gmtime` with the pointer to the current time.

❻ Create a new hash and store the pointer in `hash`.

❼ This line and those following store the contents of the structure into the hash. The first argument is a pointer to the HV, the second argument is the key and the third is the length of the key. The fourth argument must be an SV therefore an SV is created using the integer from each struct entry. The final argument is the hash number itself; since we don't know the value we pass in a `0` and ask Perl to calculate it for us. One final thing to note is that `hv_store` does not affect the reference count of the SV that is being stored. This means that each of the SV's stored in the hash will automatically have a refcount of `1`.

❽ The hash has been populated so we can copy the pointer to the RETVAL variable.

Whilst the above XS code does work, returning a reference to a hash containing the `gmtime` results, there is a subtle bug in the above code as can be illustrated by the following output:

```
% perl -Mblib -MTime -MDevel::Peek -e '$h=Time::gmtime_as_href(time);Dump($h)'
SV = RV(0x81109b4) at 0x815bd54
  REFCNT = 1
  FLAGS = (ROK)
```

```
  RV = 0x80f86e0
  SV = PVHV(0x81429a8) at 0x80f86e0
    REFCNT = 2
    FLAGS = (SHAREKEYS)
    IV = 8
    NV = 0
    ARRAY = 0x81003d8  (0:2, 1:5, 3:1)
    hash quality = 85.7%
    KEYS = 8
    FILL = 6
    MAX = 7
    RITER = -1
    EITER = 0x0
    ...
```

This shows that the reference count to the HV is 2. The variable `$h` has one of the references but there are no other variables that know about the reference. This constitutes a memory leak. If `$h` is later undefined or goes out of scope, the reference count on the HV will drop to `1` but it can't go any lower. Since it never goes to zero the HV will not be freed until the program exits. The reason for this is that when the HV is created using `newHV` its reference count is set to one as expected. The output typemap entry for an HV is

```
T_HVREF
   $arg = newRV_inc((SV*)$var);
```

which increments the reference count when the reference is taken. At this point there is an SV containing a reference to the hash on the stack and the `hash` variable containing the HV and the reference count is now, correctly, `2`. Unfortunately when the XS function exits the `hash` variable simply disappears without decrementing the reference count. Perl overcomes problems such as this by introducing the concept of *mortality*. If an SV is marked as mortal the reference count will automatically be decremented at some point later in time. For XS functions, mortal variables have their reference count decremented on exit from the function. The above code can be fixed to avoid the memory leak simply by marking `hash` as mortal with:

```
hash = (HV*)sv_2mortal((SV*)newHV());
```

The new sequence now becomes:

1. Create new HV and increment reference count. Reference count = `1`
2. Mark HV as mortal. Reference count = `1`
3. Take reference to HV and store it on the argument stack. Reference count = `2`
4. Exit XS function and automatically decrement the reference count. Reference count = `1`

If this change is made the test program now reports the correct reference count for `$h`:

```
% perl -Mblib -MTime -MDevel::Peek -e '$h=Time::gmtime_as_href(time);Dump($h)'
SV = RV(0x81109b4) at 0x815bd54
  REFCNT = 1
  FLAGS = (ROK)
```

```
  RV = 0x80f86e0
  SV = PVHV(0x81429a8) at 0x80f86e0
    REFCNT = 1
    FLAGS = (SHAREKEYS)
    IV = 8
    NV = 0
    ARRAY = 0x81003d8  (0:2, 1:5, 3:1)
    hash quality = 85.7%
    KEYS = 8
    FILL = 6
    MAX = 7
    RITER = -1
    EITER = 0x0
    Elt "yday" HASH = 0x4630b8
    SV = IV(0x81073b8) at 0x815bcdc
      REFCNT = 1
      FLAGS = (IOK,pIOK)
      IV = 77
    Elt "wday" HASH = 0x450f30
    SV = IV(0x81073b4) at 0x815bce8
      REFCNT = 1
      FLAGS = (IOK,pIOK)
      IV = 1
    Elt "mday" HASH = 0x3f6788
    SV = IV(0x81071cc) at 0x8104354
      REFCNT = 1
      FLAGS = (IOK,pIOK)
      IV = 19
```

6.5.3.2. Returning a list

An alternative way of returning a hash is to return a list with alternating keys and values:

```
%hash = Time::gmtime_as_list( time );
print "Day is ", $hash{"mday"}, "\n";
```

The XS code for this must push the keys and the values on to the argument stack just as if this was a normal Perl routine:

```
void                                                    ❶
gmtime_as_list( clock )
  time_t clock
 PREINIT:
  struct tm * tmbuf;
 PPCODE:                                                ❷
  tmbuf = gmtime( &clock );

  EXTEND(SP, 16);                                       ❸
```

```
PUSHs( sv_2mortal( newSVpv("sec", 3) ));                ❹
PUSHs( sv_2mortal( newSViv(tmbuf->tm_sec) ));
PUSHs( sv_2mortal( newSVpv("min", 3) ));
PUSHs( sv_2mortal( newSViv(tmbuf->tm_min) ));
PUSHs( sv_2mortal( newSVpv("hour", 4) ));
PUSHs( sv_2mortal( newSViv(tmbuf->tm_hour) ));
PUSHs( sv_2mortal( newSVpv("mday", 4) ));
PUSHs( sv_2mortal( newSViv(tmbuf->tm_mday) ));
PUSHs( sv_2mortal( newSVpv("mon", 3) ));
PUSHs( sv_2mortal( newSViv(tmbuf->tm_mon) ));
PUSHs( sv_2mortal( newSVpv("year", 4) ));
PUSHs( sv_2mortal( newSViv(tmbuf->tm_year) ));
PUSHs( sv_2mortal( newSVpv("wday", 4) ));
PUSHs( sv_2mortal( newSViv(tmbuf->tm_wday) ));
PUSHs( sv_2mortal( newSVpv("yday", 4) ));
PUSHs( sv_2mortal( newSViv(tmbuf->tm_yday) ));
```

❶ Here we use void as a return value for the function since the return values will be handled directly by the routine rather than by the XS compiler.

❷ We use PPCODE rather than CODE to indicate to the XS compiler that we are handling the return values ourselves. This does not affect the processing of input arguments but does imply that OUTPUT can not be used.

❸ The `EXTEND` macro makes sure that the argument stack is large enough to contain the requested number of arguments. Since we know that we will need to hold 16 items (8 keys and 8 values) we presize the stack for efficiency. This is similar to using **`$#array = 16;`** in perl. `SP` is the Stack Pointer (pointing to the current position in the stack) and is initialised for you automatically on entry to the routine.

❹ Here we need to start pushing arguments onto the stack. For XS programmers the only approved ways of pushing arguments onto the stack are the `PUSHs` and `XPUSHs` macros. These push an SV onto the argument stack. The difference is that `XPUSHs` extends the size of the stack by one so that it is guaranteed to have room for the incoming SV. In this example we could have used `XPUSHs` instead of `PUSHs` and removed the `EXTEND` call. Since we can only push SVs onto the stack each argument (string key or integer value) is first converted to an SV and then marked as a mortal. All SVs pushed on to the stack must be marked as mortal so that they can be freed after assignment. The reason for this is that a copy of the SV is assigned to a perl variable and not the original SV. If this was not the case **`$a = $b`** would alias `$a` to `$b`!

> Whilst it may be tempting to make use of the `PUSHi`, `PUSHp` and `PUSHn` functions (and the related `XPUSH`) to push plain integers, strings and floats onto the stack in XS routines they do not form part of the XS API. These routines are intended for use by the internals and can only be used to return a single value onto the stack. This is because they use a single SV (that must be declared using the `dTARG` macro) and if 5 values are pushed onto the stack with these functions they will all receive the value of the last thing that was pushed on because it is the pointer to the same SV that is stored on the stack.

6.5.3.3. Passing it back into C

Both these techniques (returning a list or returning a hash reference) overcome the memory leak problem described earlier. The reason for this is that the information is copied to a perl data structure and not kept inside a C structure. Perl knows nothing about the C structure so can only free its memory via an object destructor. When using hashes to store the information Perl can free the memory directly. Unfortunately this behaviour comes at a price when it is time to pass the information back to C. If the information is only required for information (e.g. the data returned from a call to `stat`) then this is not a problem, but if the data is to be passed back into C (e.g. to the `timegm` function) then more work is required because the data must be converted from the hash to a C structure. Even worse, you can no longer rely on the integrity of the data structure since the contents of the hash can be changed arbitrarily before they are passed back to C. If structure integrity is a problem then you should probably be using objects.

6.6. Arrays

In some cases a C routine might want to receive an array of numbers or strings. To handle this you will have to convert the perl array or list into a C array before calling the C function. This usually involves the following steps:

1. Allocating some memory to hold the array.
2. Copying each element from the list/array to the C array.
3. After the C function has run, free the memory.

6.6.1. Numeric arrays

It is quite a common requirement, especially in the scientific community, to be able to pass arrays of numbers to and from C. This section will first describe how to deal with 1-dimensional arrays and then move on to discuss multi-dimensional arrays. It will finish with some benchmarking examples to provide a guide for the most efficient handling of arrays and lists.

One additional point is that this section deals with converting the lists and arrays to C arrays and not simply manipulating a Perl array “as-is”. The easiest way to handle a Perl array in C as a Perl array is simply to pass in the array reference and manipulating the AV* in C code.

6.6.1.1. into C

In this section we will illustrate how to pass Perl numeric arrays to C by providing an XS interface to a function that will sum the elements of the array and return the answer to Perl. The signature of the C function will be:

```
int sum(int count, intArray * array);
```

where intArray is typedef'ed to an int so that XS can distinguish a pointer to an integer from a pointer to an array of integers (they are both written as int * in C). For this example the module will be called `Arrays`. Here is the top of the XS file including the `sum` function:

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

typedef int intArray;

/* Add up everything in an int array */
/* Args: the number of things to add, pointer to array */

int sum ( int num, intArray * array ) {
  int thesum = 0;
  int count;
  for (count = 0; count < num; count++) {
    thesum += array[count];
  }
  return thesum;
}

MODULE = Arrays PACKAGE = Arrays
```

6.6.1.1.1. ...as a list

One of the simplest ways to pass an array into XS (and into other Perl subroutines) is simply to pass in a list:

```
$sum = Arrays::sum_as_list( @values );
```

Here each element of the array `@values` is pushed onto the argument stack so in XS each argument must be retrieved from the stack and copied into a C array. Perl provides the T_ARRAY typemap entry for handling this situation:

```
int
sum_as_list( array, ...)
  intArray * array
 CODE:
  /* ix_array is the total number of elements */
  RETVAL = sum( ix_array, array );
 OUTPUT:
  RETVAL
 CLEANUP:
  Safefree( array );
```

This all looks fairly straightforward but there are many things going on behind the scenes:

- T_ARRAY is unique in the standard typemap file in that it is designed to work on multiple input arguments. The ellipsis (...) is used to indicate an unknown number of arguments are expected but they are all processed as part of the typemap entry.
- T_ARRAY is greedy. Only a single XS argument can be associated with T_ARRAY and it must be the last argument in the list. This should not be surprising since it is doing the C equivalent of **`@args = @_;`** There can be arguments before the final list.
- T_ARRAY creates a variable called `ix_${var}` (in our example this is `ix_array`) that contains the number of elements processed by T_ARRAY.
- Memory must be allocated in which to store the new integer array. T_ARRAY assumes that there is a memory allocation function called `$ntype` (in this case `intArrayPtr`) that will return a pointer to some memory. It is passed a single argument containing the number of elements in the array. Of course, this memory must be freed at the end of the XS function.
- The XS compiler works out how to copy elements from the Perl list into the C array by guessing the C type of the variables from the type of the array. It does this by removing any mention of "Array" and "Ptr" from the `$ntype` variable and then looking in the standard typemap entry for the resulting string. In this example `$ntype` is intArrayPtr so each element is copied using the int typemap entry.

For completeness, here is the typemap entry for T_ARRAY:

```
U32 ix_$var = $argoff;                              ❶
$var = $ntype(items -= $argoff);                    ❷
while (items--) {                                   ❸
    DO_ARRAY_ELEM;                                  ❹
    ix_$var++;
}
/* this is the number of elements in the array */
ix_$var -= $argoff                                  ❺
```

❶ Declare a new variable and set it initially to the position of the first element in the list. This declaration does cause problems if previous arguments have used complicated typemaps themselves. This is because C does not like variable declarations part way through a block. This issue is discussed further in Section 6.10

❷ Allocate some memory using function `$ntype`. The requested number of elements is calculated by using (and modifying) `items`. XS automatically sets this variable to the total number of XS arguments.

❸ Loop over each element until there are no more remaining. `items` is decremented until it hits zero whilst `ix_$var` is incremented to provide an index into the argument list.

❹ DO_ARRAY_ELEM is the magic used by the XS compiler to indicate that an element must be copied from the stack to `$var`. It uses `ix_$var` to index into the stack and derives the type of the element from the type of the array.

❺ Finally, reset the value of `ix_$var` so that it now reflects the number of elements in the C array.

So far we have not said much about the memory allocation function in the current example. The default allocator (implied by the use of the `Safefree` function in the above example) could look something like this:

```
intArray * intArrayPtr ( int num ) {
  intArray * array;

  New(0,array, num, intArray );
  return array;
}
```

where we simply use the `New` macro to allocate `num` integers and return the pointer. Whilst this will work we still have to make sure that the memory is freed when we have finished with it. In most cases this just involves the use of a CLEANUP block in the XS definition since we usually don't want to keep the memory. Since laziness is sometimes a virtue there is another approach to memory allocation which will obviate the need for the CLEANUP block. During the discussion of structure handling we introduced the concept of mortality. Perl uses mortal variables to make sure that variables are automatically freed when a Perl scope is exited. We can make use of this fact by allocating memory in a Perl scalar, marking it as mortal and then letting perl free the memory when the XS function is completed. The memory allocation function then becomes:

```
intArray * intArrayPtr ( int num ) {
  SV * mortal;
  mortal = sv_2mortal( NEWSV(0, num * sizeof(int) ) );
  return (intArray *)SvPVX(mortal);
}
```

This function creates a new SV and makes sure that the PV part of the SV is large enough to hold the required number of integers (the `sizeof` function is used to determine how many bytes are required for each integer). This SV is marked as mortal and the pointer to the PV part is returned using the `SvPVX` macro. If this function is used the CLEANUP section of `array_as_list` can then be removed.

If we place this memory allocation function before the XS code and remove the CLEANUP section this example, once built, will sum up all elements in an array:

```
% perl -Mblib -MArrays -e 'print Arrays::sum_as_list(5,6,7)'
18
```

* *Do we need to show the complete example again?*

6.6.1.1.2. ...as an array reference

Just like when programming in Perl, an alternative to passing in a list is to pass in a reference to an array:

```
$sum = Arrays::sum_as_ref( \@values );
```

The main advantage of this technique is that multiple arrays can be passed to a function. The XS code for this is as follows:

```
int
sum_as_ref( avref )
  AV * avref;                                          ❶
 PREINIT:
  int len;
  int i;
  SV ** elem;                                          ❷
  intArray * array;
 CODE:
  len = av_len( avref ) + 1;                           ❸
  array = intArrayPtr( len );                          ❹

  /* copy numbers from Perl array */
  for (i=0; i<len; i++) {                              ❺
    elem = av_fetch( avref, i, 0);                     ❻
    if (elem == NULL) {                                ❼
      array[i] = 0;
    } else {
      array[i] = SvIV( *elem );                        ❽
    }
  }
  RETVAL = sum( len, array );                          ❾
 OUTPUT:
  RETVAL
```

❶ The argument is a pointer to an AV. The default typemap entry will make sure that we have an array reference and will exit the program if we don't get one.

❷ Declare `elem` as a pointer to a pointer to an SV. This is the type of variable returned by `av_fetch`.

❸ Find out how many elements are in the array.

❹ Allocate some memory for the C array using the same function that we used for `sum_as_list`.

❺ Loop over each element in the AV, copying it to the C array.

❻ Retrieve the i'th element from the array.

❼ A complication here is that `av_fetch` can return NULL for the requested element. This means that we first have to check that the pointer is valid before dereferencing it.

❽ Copy the integer part of the SV to the C array. Since `SvIV` expects a pointer to an SV we must dereference `elem`.

❾ Finally run the `sum` function.

Once built this should produce the same answer as the previous example but this time an array reference is used:

```
% perl -Mblib -MArrays -e 'print Arrays::sum_as_ref([5,6,7])'
18
```

6.6.1.1.3. ...as a packed string

> In XS the sense of pack/unpack and input/output are very different to that expected by a Perl programmer. INPUT is used in XS to indicate data passing into C (and not out of Perl) and OUTPUT is used to indicate data passing out of C and into Perl. More confusing is that occassionally you will see the term "pack" used to indicate conversion of a C array to a Perl array and "unpack" to indicate conversion of a Perl array to a C array. This is completely different to the Perl use of the `pack` and `unpack` functions and we will use the Perl sense in this chapter. This means that a C array is a packed form of a perl array. This makes sense since a C array uses less memory than a perl array. These confusions arise because XS exists so that Perl data can be handled by the Perl internals and therefore the internals are seen as the primary consumer of the data.

The third way to pass an array into C is to pack the perl array into a byte string and then pass that string into C where it will be treated as a C array. From perl this would look like the following:

```
$packed = pack("i*", @values);
$sum = Arrays::sum_as_packed( $packed );
```

In XS you could implement this as follows:

```
int
sum_as_packed( packed )
  SV * packed                                          ❶
 PREINIT:
  int len;
  intArray * array;
 CODE:
  array = (intArray *)SvPV( packed, PL_na );           ❷
  len = SvCUR( packed ) / sizeof(intArray);            ❸
  RETVAL = sum( len, array );
 OUTPUT:
  RETVAL
```

❶ We want to interrogate the SV directly rather than extracting a specific piece of information.

❷ Retrieve a pointer to the byte array from the SV

❸ Calculate the number of elements in the array by asking the SV for the total number of bytes and then dividing by the number of bytes used to represent an integer.

The main point here is that we are using the SV directly rather than asking XS to translate it for us. This is useful since the SV knows how many bytes it is holding. If this information is not required or if you can pass in the number of elements of the array as an argument[10] this XS code can be simplified:

```
int
sum_as_packed2( len, packed )                          ❶
  int len
  char * packed                                        ❷
 CODE:
  RETVAL = sum( len, (intArray *)packed );             ❸
 OUTPUT:
  RETVAL
```

❶ Now the length of the array is included as an argument.

❷ Use char * to indicate that we are interested in the PV part of the SV. Alternatively we could have associated intArray * with T_PV in the typemap file.

❸ Since we have a pointer to a char we have to cast the pointer to type intArray before passing it to the `sum` function.

6.6.1.2. out of C

Arrays can be returned to Perl as either a list pushed onto the stack or by creating an array and returning the reference. We saw in Section 6.5.3 how to return a hash reference and a list and the only difference for arrays is the use of `av_` functions rather than `hv_` functions.

This section will highlight some additional methods for returning (numeric) arrays which may be useful. We put these here for completeness, since they are used in existing code, but they are probably not the best approach for new code.

6.6.1.2.1. ...as a list without PPCODE

The T_ARRAY typemap entry was used in Section 6.6.1.1.1 to pass a list into C and it can, in principal, also be used to return a list from C without having to worry about looping and using the `PUSHs` macro (see Section 6.5.3 for details on pushing elements onto the return stack).

The main problem with this is that it only works with XS CODE blocks (since OUTPUT typemap entries are only used when the OUTPUT keyword is used in XS) but the XS compiler always forces a single return value. In general it is safer to ignore T_ARRAY for output and just use PPCODE instead.

If you do want to use this then the easiest trick is simply to co-opt the CLEANUP section and make explicit use of the `XSRETURN` function. Here is an example of how to use T_ARRAY to return an array of integers:

```
intArray *                                          ❶
test_t_array()
 PREINIT:
  intArray test[2];                                 ❷
  U32 size_RETVAL;                                  ❸
 CODE:
  test[0] = 1; test[1] = 2;                         ❹
  size_RETVAL = 2;                                  ❺
  RETVAL = test;                                    ❻
 OUTPUT:
  RETVAL                                            ❼
 CLEANUP:
  XSRETURN(size_RETVAL);                            ❽
```

❶ The return type is now a pointer to an array. This uses the same typemap we have used for the previous array examples.

❷ Create a test array in C to contain 2 elements.

❸ T_ARRAY requires the declaration of this variable (technically declared as `size_$var` in the typemap file but this variable will almost always be associated with `RETVAL`). It is used by the typemap to determine how many elements in the array are to be copied to the stack.

❹ For this example simply copy two numbers into the array.

❺ Store the size of the array.

❻ `RETVAL` now points to the first element of our test array.

❼ Mark `RETVAL` for output.

❽ This macro will exit the XS routine just before the normal exit provided by **xsubpp**. The argument indicates how many items have been placed on the return stack.

6.6.1.2.2. ...as a packed string

Just as it is possible to pass to XS a byte array generated by the Perl `pack` function (see Section 6.6.1.1.3) it is also possible to return a byte array that can be unpacked with the Perl `unpack` function. If you know how many elements are to be stored in the array at compile time XS provides a way or returning the packed string to Perl. This example returns 3 integers as a packed string:

```
array(int, 3)
return_packed()
 PREINIT:
  intArray test[3];
 CODE:
```

```
  test[0] = 1; test[1] = 2; test[2] = 3;
  RETVAL = test;
 OUTPUT:
  RETVAL
```

When compiled this code copies **`3 x sizeof(int)`** bytes from `RETVAL`. They can be unpacked in perl with **`unpack("i*",$retval)`**.

If the size of the return array is not known at compile-time the bytes must be copied to the perl variable using a modified form of the T_OPAQUEPTR typemap entry:

```
intArray *
return_npacked()
 PREINIT:
  U32 size_RETVAL;
  intArray test[3];
 CODE:
  test[0] = 1; test[1] = 2; test[2] = 3;
  size_RETVAL = 3;
  RETVAL = test;
 OUTPUT:
  RETVAL
```

with a corresponding typemap entry of:

```
intArray * T_OPAQUEARRAY

OUTPUT
T_OPAQUEARRAY
sv_setpvn($arg, (char *)$var, size_$var * sizeof(*$var));
```

Here we associate intArray * with T_OPAQUEARRAY. The only difference between this and T_OPAQUEPTR is that we have used the `size_$var` variable to indicate how many elements to copy.

In general, if packed strings are returned and the bytes are not required directly, it is usually better to provide a perl wrapper to the XS function so that the bytes are hidden from the caller.

> When looking through the typemap file you will see entries called T_PACKED and T_PACKEDARRAY. These are not designed for dealing with packed strings!

6.6.1.3. The Perl Data Language

If you are dealing with large or multi-dimensional arrays the techniques described so far will probably prove inadequate and you should seriously consider changing approach and using PDL. The Perl Data Language (PDL) was developed as a means to handle multi-dimensional arrays in perl compactly and efficiently. These three issues are extremely important in scientific computing and image processing for the following reasons:

multi-dimensionality

Perl has no real concept of multi-dimensional arrays. An array can have references to other arrays in order to simulate additional dimensions but there is nothing in the language to force the same number of elements in each row or column. When the dimensionality is greater than two the perl approach becomes unwieldy and it is very time consuming to check the dimensionality of the data. The following code shows how perl handles a 1-, 2- and 3-d array:

```
@oned   = ( 1, 2 );
@twod   = ( [1,2], [3,4] );
@threed = ( [ [1,2], [3,4] ],  [ [5,6],[7,8] ] );
```

compactness

When the number of elements in array is large, the representation of that array can have an enormous effect on the memory requirements of the program. In Section 4.3 we saw that both Perl arrays and Perl scalars have a significant memory overhead compared to that required for single numbers but that this is accepted because of the enormous gain in functionality this brings. In situations where we are dealing with blocks of numbers this flexibility is not required. As an example, in the 3-d array above, this requires 7 perl arrays, 8 integer scalars and 6 references. On a 32-bit linux system this is about 128 bytes for just 8 numbers (assuming 12 bytes for sv_any, 16 bytes for a xpviv and 44 bytes for a xpvav). A C representation will require just 32 bytes (8 elements of 4 bytes each). Clearly for arrays of a million pixels the closer the representation is to pure C the more significant will be the saving on memory.

speed

So far in this section we have shown that passing arrays into an out of perl requires loops to pack and unpack the array each time. In cases where a large data array is being passed continually to and from C the time overhead in doing this will be enormous. Additionally, for N-dimensional data arrays the large number of dereferences required to return the data values will be significant.

It is therefore not surprising that PDL was developed by scientists[11] as an attempt to solve these problems without having to use expensive proprietary packages or a language other than Perl!

PDL deals with the problems of dimensionality, speed and compactness by using a PDL object (known as a “piddle”[12]) to store the data itself and information such as the dimensionality and data type. In reality a piddle is a C struct for efficiency and the data is stored as a normal C array[13]. We will not attempt to be a complete guide to PDL in this book but more importantly for this book, we will show how to interact with PDL from within XS.

6.6.1.3.1. A primer

Rather than try to cover all of PDL here is a quick introduction. PDL provides a shell for interactive use (called **perldl**) which can be very useful for general experimentation with Perl as well as for PDL. Here are some examples that will hopefully provide a taster (they can be typed in at the **perldl** shell or in a program in conjunction with `use PDL;`):

$a = pdl([0,1,2],[3,4,5]);

Create a 3x2 piddle

$a = sequence(3,2);

Create the same piddle using the `sequence` command. This is effectively an n-dimensional version of the `..` Perl operator.

print $a;

Print a stringified form of the piddle. This works for reasonably sized piddles. Here the output is:

```
[
 [0 1 2]
 [3 4 5]
]
```

$a *= 2;

Multiply each element in the piddle by `2`. `$a` is now:

```
[
 [ 0  2  4]
 [ 6  8 10]
]
```

In PDL the standard operators (+,-,*,/ etc) are overloaded so that piddles act like normal perl variables. By default PDL does not do matrix operations on piddles (but it can be made to).

$b = $a->slice("1,");

Extract a slice from `$a`. Here we use object notation to invoke the slice method. In this case we are extracting column 1:

```
[
 [2]
 [8]
]
```

$c = pdl(10,20); $b += $c;

Create a 2 element piddle and add it to `$b` which becomes:

```
[
 [32]
 [38]
]
```

More importantly, `$a` now becomes:

```
[
 [ 0 32  4]
 [ 6 38 10]
]
```

because `$b` is still related to `$a` so that changes in one are reflected in the other. This is one of the most powerful features of PDL.

PDL is a powerful tool for manipulating array data and should be considered seriously for any project that is dealing with arrays and perl.

6.6.1.3.2. PDL and XS

Now that we have shown the utility of PDL when using arrays we will now show how to use PDL from XS. When viewed from Perl a PDL is seen as an object but from within C a PDL is represented as a structure (denoted by a pdl *). In general you should only use PDL with XS if you want direct access to the structure. PDL provides easier methods of passing data to C routines with the `PDL::PP` (more details of which can be found in Section 7.4) and `PDL::CallExt` modules.

The PDL infrastructure provides typemap entries to handle the conversion from/to the PDL structure:

```
pdl *                   T_PDL

INPUT

T_PDL
        $var = PDL->SvPDLV($arg)

OUTPUT

T_PDL
        PDL->SetSV_PDL($arg,$var);
```

These typemap entries use the programming interface (API) provided by the PDL core to translate perl objects to the PDL structures. This raises the issue of how to use C functions that are provided by a separate perl module in your XS code. For external C libraries you simply make sure that you link

against the library when the XS module is built. The PDL shared library[14] is installed somewhere in the Perl site library tree in `PDL/Core/Core.so` (on many Unix systems). There are a number of difficulties associated with attempting to use this library directly from other XS modules:

1. In order to link your PDL XS code against this library you would first need to locate it (using the `Config` module) and then convince MakeMaker that it should be included even though it does not look like a standard library (this can be done by fooling MakeMaker into thinking that `Core.so` is an object file).
2. Each PDL-based XS module will have to jump through the same hoops.

To simplify access to the PDL API, pointers to the public functions are stored in a C structure. A pointer to this structure is then stored in a Perl variable in the PDL namespace. In order to use a PDL function all that is required is to retrieve this pointer from the Perl variable. This is the approach also taken by the perl/Tk module. XS provides a means of doing this at load time using the BOOT: section and PDL recommends the following code:

```
#include "pdl.h"
#include "pdlcore.h"                                    ❶
static Core * PDL;                                      ❷
SV* CoreSV;                                             ❸

MODULE = XXXX   PACKAGE = XXXX

BOOT:
   /* Get pointer to structure of core shared C routines */
   CoreSV = perl_get_sv("PDL::SHARE",FALSE);  /* SV* value */❹
   if (CoreSV==NULL)                                    ❺
     Perl_croak(aTHX_ "This module requires use of PDL::Core first");
   PDL = (Core*) (void*) SvIV( CoreSV );  /* Core* value */❻
```

❶ Standard include files for PDL. These declare the PDL constants and the function structure.

❷ Declare a pointer to a Core. A Core is typedeffed to struct Core in `pdlcore.h`. It is declared static since we want to retain the value each time we call a function.

❸ A pointer to an SV. This is used to store the SV retrieved from the PDL namespace.

❹ Get the variable `PDL::SHARE`

❺ We must load the `PDL::Core` module before attempting to load this module else `PDL::SHARE` will not be defined. This is achieved most easily simply by making sure that our perl module loads `PDL::Core` before calling the bootstrap method.

❻ Retrieve the integer part of the SV, cast it to type Core* and store it in a C variable called `PDL`.

Now we can see that the standard PDL typemap entries assume that we have done the above because they use the variable `PDL` as a pointer to a structure in order to run the conversion methods. Now we can use any public PDL function simply by using this variable.

With this ground work in place we can now write a PDL version of our routine to sum the elements in an array using the `sum` function presented earlier[15]. Here is the XS snippet:

```
int
sum_as_pdl( in )
  pdl * in                                                    ❶
 CODE:
  PDL->converttype( &in, PDL_L, 1);                           ❷
  RETVAL = sum( in->nvals, (intArray *)in->data);             ❸
 OUTPUT:
  RETVAL
```

❶ Here we assume that we have a PDL argument. If an array is passed in it will be rejected by `SvPDLV`.

❷ A PDL is typed (by default all PDLs are double precision) and this line is responsible for converting it to an integer type so that it can be summed by our function. This example is slightly naughty because it converts the input PDL to integer format. In a real application either a copy should be made so that the input piddle is not modified or a `sum` should be written for each data type.

❸ Here we find the number of elements in the PDL using the `nvals` part of the structure and retrieve the values themselves using the `data` part.

In order to compile this XS code we need to generate a perl module wrapper that will load PDL itself and a `Makefile.PL` that will correctly locate the PDL include files from the installed tree. Here is a suitable `pm` file:

```
package Arrays;

use PDL::Core;
use base qw/ DynaLoader /;

our $VERSION;

$VERSION = '1.00';

bootstrap Arrays;
```

and corresponding `Makefile.PL`:

```
use ExtUtils::MakeMaker;
use File::Spec;
use Config;

my $pdlinc = '-I'.
             File::Spec->catdir($Config{'installsitearch'},
                                "PDL", "Core");               ❶

# Write the makefile
```

```
WriteMakefile(
              'NAME'=> 'Arrays',
              'VERSION_FROM' => 'Arrays.pm', # finds $VERSION
              'PREREQ_PM' => { 'PDL' => '2.0'},           ❷
              'INC'       => $pdlinc,
             );
```

❶ The PDL-specific include files (`pdl.h` and `pdlcore.h`) are installed as part of PDL into the perl `installsitearch` directory. This line uses the `Config` module to determine that location and `File::Spec` to append the PDL directory to that location.

❷ The PDL internals were completely rewritten for version 2.0. This line instructs MakeMaker to check the version of the installed PDL and to complain if the version number is less than 2.0.

6.6.1.4. Benchmarks

So far we have shown 4 ways of passing numeric data into C for processing and the associated ways of returning arrays back to Perl. These methods can be summarised as:

1. Using a list.
2. Using a reference to an array.
3. Using a packed string.
4. Using a PDL object.

To finish off this section on arrays we will now write a simple benchmark to compare the efficiency of these techniques using the summing code described earlier with the exception that we will use the native PDL `sum` function.

```
use Benchmark;
use Arrays;
use PDL;
use strict;

my @array = (0..100);
my $pdl  = sequence(long,101);

timethese(-3, {
       'PDL' => sub { sum($pdl); },
       'List'=> sub { Arrays::sum_as_list(@array) },
       'Ref' => sub { Arrays::sum_as_ref(\@array) },
       'Pack'=> sub { Arrays::sum_as_packed( pack("i*", @array) ); },
      })
```

The above benchmark runs for at least 3 seconds and gives the following output:

```
Benchmark: running List, PDL, Pack, Ref, each for at least 3 CPU seconds...
      List:  3 wallclock secs ( 3.28 usr +  0.00 sys =  3.28 CPU) @ 110633.84/s (n=362879)
       Ref:  2 wallclock secs ( 3.01 usr +  0.00 sys =  3.01 CPU) @ 77112.62/s (n=232109)
      Pack:  4 wallclock secs ( 3.34 usr +  0.00 sys =  3.34 CPU) @ 52336.53/s (n=174804)
       PDL:  4 wallclock secs ( 3.08 usr +  0.00 sys =  3.08 CPU) @ 10284.09/s (n=31675)
```

Since we are asking `Benchmark` to run for a specific amount of time the important numbers are in the last but one column; the number of times the subroutine was executed per second. This column indicates that for a small array (in this case 101 elements) a list is ten times faster than a PDL, twice as fast as using pack and one and a half times faster than using a reference. The PDL solution is surprisingly slow in this case but this is in part due to the additional overhead that is present in the PDL system but which is not being used by our example. The packed string is expected to be slow since it calls an additional Perl function each time. The reference is slower than list due to the overhead of taking the reference. If we now increase the size of the array by two orders of magnitude to 10000 elements we get a different result:

```
Benchmark: running List, PDL, Pack, Ref, each for at least 3 CPU seconds...
      List:  3 wallclock secs ( 3.20 usr +  0.02 sys =  3.22 CPU) @ 1495.65/s (n=4816)
       PDL:  4 wallclock secs ( 3.20 usr +  0.00 sys =  3.20 CPU) @ 4372.81/s (n=13993)
      Pack:  3 wallclock secs ( 3.14 usr +  0.00 sys =  3.14 CPU) @ 448.09/s (n=1407)
       Ref:  3 wallclock secs ( 3.08 usr +  0.00 sys =  3.08 CPU) @ 917.21/s (n=2825)
```

Now PDL is much faster than the rest so that the overhead due to the PDL infrastructure is now becoming insignificant when compared to the cost of converting large arrays into C data structures.

Of course, specific benchmarks can not tell the whole story and the final choice you make depends on many different factors. For example, if you require multiple array arguments then you can not use a simple list, or if you want maximum portability you may not want to insist on the availability of PDL.

6.6.2. Character strings

We saw in Section 3.3.1 that an array of strings is represented in C as an array of pointers that point to the memory location of the start of each string; a string is an array of characters and a string array is an array of pointers. In general for XS the char** can be created and populated just as for any other array although care must be taken if the new array is to persist after the call to the XS function. This is because in the simple case, you simply copy the pointers from the SV's and use them but if you return to Perl you can not guarantee that the SV will still be around at a later date. In that case you will have to take copies of the entire string rather than just storing the pointer.

Converting a char** to a Perl array is simply a case of stepping through the C array and copying the contents to the Perl array.

The following XS code demonstrates both techniques by copying an input array to a char** and then copying that back onto the output stack. Variants involving output references or input lists will be very similar and this example does not have complete error checking.

```
void
copy_char_arr( avref )
   AV * avref;                                          ❶
  PREINIT:
   char ** array;
   int len;
   SV ** elem;
   int i;
  PPCODE:
   len = av_len( avref ) + 1;
   /* First allocate some memory for the pointers */
   array = (char**) get_mortalspace( sizeof(void *) * len );   ❷

   /* Loop over each element copying pointers to the new array */
   for (i=0; i<len; i++) {
     elem = av_fetch( avref, i, 0);
     array[i] = SvPV( *elem, PL_na );                   ❸
   }

   /* Now copy it back onto the stack */
   for (i=0; i<len; i++) {
     XPUSHs( sv_2mortal( newSVpv( array[i], 0)));       ❹
   }
```

❶ In this example the input is expected to be a reference to an array and the output is a list:

```
@copy = copy_char_arr(\@src);
```

❷ Get some temporary storage to hold the array of pointers. The `get_mortalspace` is identical to the `intArrayPtr` function shown earlier except that it takes the number of bytes as argument rather than the number of integers.

❸ Retrieve the pointer from the SV (converting it to a PV if required) and then store it in the array. If it was necessary to copy the string first we would also need to allocate some memory for it here.

❹ Copy each string from the string array back into a new SV and push it onto the argument stack.

6.7. Callbacks

A *callback* is the name given to a user supplied function that is called by another function. The classic example of callbacks in perl is the `Tk` module. Whenever a Tk event occurs (for example, a button is pushed on the GUI) Tk sees whether a perl subroutine should be called to process the event. The following code shows how a Tk callback can be set up:

Example 6-1. A simple `Tk` callback

```
use Tk;
use strict;

my $MW = new MainWindow;
my $button = $MW->Button( -text => "Exit",
                          -command => sub { exit; } );
$button->pack();
MainLoop;
```

If this program is run it will put up a window containing a single button. If the button is pressed the callback associated with the button (configured using the `-command` option) will be executed and the program will therefore exit. The callback is not called directly by user code, it is called from the event loop from C code.

The main difficulty handling callbacks in XS is that perl stores subroutines in a CV whereas C callbacks are implemented as pointers to a C function. In order for C to call a perl subroutine an intermediate function has to be inserted that knows about Perl. This indirection leads to all the complications associated with using callbacks from perl.

In general there are usually three types of callbacks that must be handled:

- A callback that is used for a single command with return passing back to perl once the callback has been used. This is common in the POSIX `qsort` function and the unix search functoins.
- A single callback that is registered and one point in the program and then executed some time later (for example an error handler)
- Multiple subroutines registered as callbacks that can be called at any time (event driven programs are examples of this)

We will cover each of these types in turn.

6.7.1. Immediate Callbacks

The simplest type of callback is one where the C function executes given the supplied callback and then completes before returning back from XS. The C `qsort` function provides an excellent example of this. This function can be used to sort arrays and used to be what Perl used to implement the Perl `sort` routine. The calling signature is:

```
void qsort(void *base, size_t nel, size_t width, void *compar);
```

where *`base`* is a pointer to the start of the array, *`nel`* is the number of elements in the array, *`width`* is the number of bytes used to represent each element and *`compar`* is a pointer to a function that is used to

compare individual elements of the array. It is the *`compar`* function that holds the C callback. Obviously the Perl interface to this function should behave like the standard `sort` function, that is:

```
@sorted = qsorti \&compar, @unsorted;
```

We are calling the function `qsorti` to indicate that this sort function can only be used to sort arrays of integers. This simplifies the example code and allows us to focus on the implementation of the callback rather than the complication of handling all data types. For this examplee we are going to use a module called `CallBack`. The following XS code implements the `qsorti` function:

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

typedef int intArray;                                          ❶


/* Static memory for qsort callback */
static SV * qsortsv;                                           ❷

/* Routine to allocate memory for integer array */
/* Allocate the memory as a mortal SV so that it is freed automatically */
intArray * intArrayPtr ( int num ) {                           ❸
  SV * mortal;
  mortal = sv_2mortal( NEWSV(0, num * sizeof(intArray) ) );
  return (intArray *)SvPVX(mortal);
}

/* The callback for qsort */
int qsorti_cb( const void *a, const void *b) {                 ❹

  dSP;                                                         ❺
  int count;
  int answer;

  ENTER;                                                       ❻
  SAVETMPS;
  PUSHMARK(SP);

  /* Push some SVs onto the stack with the values of a and b */
  XPUSHs(sv_2mortal(newSViv(*(int *)a)));                      ❼
  XPUSHs(sv_2mortal(newSViv(*(int *)b)));

  PUTBACK;                                                     ❽

  count = call_sv(qsortsv, G_SCALAR );                         ❾

  SPAGAIN;                                                     (10)

  if (count != 1)                                              (11)
      croak("User defined qsort callback returned more than 1 value\n");
```

```
  answer = POPi;                                              (12)

  FREETMPS;                                                   (13)
  LEAVE;

  return answer;                                              (14)
}

MODULE = CallBack  PACKAGE = CallBack                         (15)

void
qsorti(cb, array, ...)                                        (16)
  SV * cb                                                     (17)
  intArray * array
 PREINIT:
  U32 i;
 PROTOTYPE: &@                                                (18)
 PPCODE:
  qsortsv = cb;                                               (19)
  qsort( array, ix_array, sizeof(int), qsorti_cb);            (20)

  /* now need to push the elements back onto the stack */
  for ( i =0; i < ix_array; i++) {                            (21)
    XPUSHs(sv_2mortal(newSViv(array[i])));
  }
```

❶ We create a new type based on int so that we can associate it with the T_ARRAY typemap.

❷ This is some static memory that is used to store the code reference. This is required because our C callback has to have access to the Perl code reference.

❸ This is called automatically as part of the T_ARRAY typemap entry. It is used to dynamically allocate memory for the C integer array. It uses a mortal SV to allocate the memory rather than the `New` macro. Doing this allows us to not worry about freeing the memory since Perl will do it automatically when the XS function returns.

❹ The C function that is called by `qsort`

❺ Declare variables that we need. `dSP` is just a macro that gives us access to Perl's argument stack.

❻ `ENTER`, `SAVETMPS` and `PUSHMARK` are always used on entry to a callback to allow perl to store the current status of the stack. These are paired with `PUTBACK`, `FREETMPS` and `LEAVE`

❼ Push the arguments supplied by `qsort` on to the stack so that our perl callback can access them.

❽ This indicates that we have finished configuring the stack for our Perl function. It forms the end bracket for the `PUSHMARK`

❾ This calls the perl code block contained in `qsortsv` and returns the number of arguments that were placed onto the stack by the subroutine. The G_SCALAR flag indicates that we are calling the code block in a scalar context.

(10) Since the stack no longer reflects the state it was in on entry (because the perl subroutine we just called has messed with it) we use `SPAGAIN` to reset it.

(11) In the unlikely event that more than one argument was returned from the perl subroutine we shut down the program. This should not happen because we forced scalar context.

(12) Read the answer (as an integer) off the stack.

(13) Free any temporary SVs that were created (for example the two that we pushed onto the stack) and leave the current context.

(14) Return the answer to `qsort`

(15) Begin the XS code.

(16) Our XS function is called `qsorti` and it takes a code reference and variable length list as arguments. The list will be processed using the T_ARRAY typemap entry.

(17) We use a simple SV as argument since the `call_sv` function can automatically deal with an SV containing a reference to a CV.

(18) Specify a prototype for the XS function. This prototype matches that of the Perl `sort` function.

(19) Here we store the code reference in a static variable for later retrieval by our C callback.

(20) Run the normal C `qsort` function using our array and the C callback that we defined at the top of the file. The `ix_array` variable is defined and set by the T_ARRAY typemap.

(21) Finally we unpack the integer array and push it onto the return argument stack.

We also need a private `typemap` to indicate that an intArray * should be processed using the T_ARRAY typemap entry:

```
intArray * T_ARRAY
```

Once this module is compiled you will be able to do the following:

```
use strict;
use CallBack;

my @unsorted = (20,4,6,5,10,1);

my @sorted =  CallBack::qsorti { $_[0] <=> $_[1] } @unsorted;
print join("-",@sorted);
```

The only differences between this and the normal `sort` function are that it only sorts integer arrays and there is no special use of `$a` and `$b`. This means that `@_` must be used to obtain the sort arguments.

To summarize, we had to do the following to use the callback:

1. Write a C function that can be used as the callback and then use this function to configure the stack and call the perl subroutine.
2. Store the code reference (as an SV*) in some static memory for later retrieval.
3. Call the C function (in this case `qsort`) using our intermediary C function as the callback.

6.7.2. Deferred Callbacks

A deferred callback is one where the callback is registered with one command but called from another later in the program. A common example of this is an error handler. The error handler is registered early on in the program but it is only called when an error occurs.

To demonstrate this usage we will use our existing `qsort` example but change the Perl calling interface to:

```
void register_qsort_cb(\&callback);


@sorted = qsorti_cb(@unsorted);
```

The obvious implementation of this is to simply split the existing XSUB entry into two parts:

```
void
register_qsort_cb( cb )
  SV * cb
 CODE:
  qsortsv = (SV *) cb;

void
qsorti_cb(array, ...)
  intArray * array
 PREINIT:
  U32 i;
 PPCODE:
  qsort( array, ix_array, sizeof(int), qsorti_cb);

  /* now need to push the elements back onto the stack */
  for ( i =0; i < ix_array; i++) {
    XPUSHs(sv_2mortal(newSViv(array[i])));
  }
```

The correspondingly modified test program would be:

```
use strict;
use CallBack;

CallBack::register_qsort_cb( sub {$_[0] <=> $_[1] } );
my @unsorted = (20,4,6,5,10,1);
my @sorted = CallBack::qsorti_cb( @unsorted );
print join("-",@sorted);
```

If we run this we get the following:

```
Undefined subroutine &main::20 called at ./cbtest line 6
```

(or something similar depending on your system). What is happening here? The problem is that the SV stored in `qsortsv` has been reallocated by perl between registering it and using it. Specifically in this case the SV now seems to be holding the first element of the new array. Since we are only storing a pointer to an SV that is meant to contain a reference to a subroutine (or a name of a sub) we are sensitive to the SV changing. To overcome this problem we can either copy the contents of the SV to a new SV when storing the callback or extract the CV that is referenced by the SV and store a pointer to that instead. Changing `register_qsort_cb` to:

```
void
register_qsort_cb( cb )
  CV * cb
 CODE:
  qsortsv = (SV *) cb;
```

fixes the problem since the CV * typemap entry retrieves the reference from the SV. One issue is that this will only work with code references (rather than sub names) but usually that is not a problem. A bigger problem is that technically the reference count on the CV should be incremented when it is stored to indicate to Perl that another part of the system is interested in the CV. This also means that the reference count should be decremented on the old CV whenever a new callback is registered. For simple systems this is usually not worth bothering with (in most cases you can be assured that some other part of the system will be keeping the CV alive!) but the better solution is to simply copy the SV on entry:

```
void
register_qsort_cb( cb )
  SV * cb
 CODE:
  if (qsortsv == (SV*)NULL) {
      /* This is first time in so create an SV */
      qsortsv = newSVsv(cb) ;
  } else {
      /* overwrite since we have already stored something */
      SvSetSV(qsortsv, cb) ;
  }
```

since the `SvSetSV` function (`newSVsv` also calls it) automatically takes care of reference counting. This method relies on knowing whether the function has been called before so to guarantee this the declaration of `qsortsv` must be modified slightly to be more explicit:

```
static SV * qsortsv = (SV*)NULL;
```

6.7.3. Multiple Callbacks

So far we have only registered a single callback at any one time and that callback has been stored in a static variable. In more complex situations, such as event driven programs, we would like to store many

callbacks so a different scheme must be used; currently each time a callback is registered the previous callback is lost. There are a number of approaches to this and the choice depends mainly on how the underlying library is implemented.

1. If the calling interface provides a means for storing extra information along with the C callback (for example a C struct is used that can contain a pointer to user-supplied data) then store a copy of the SV in there and retrieve it when the callback occurs. This is the approach taken by the `Tk` module.
2. If there is no provision for user-data but the callback is associated with a specific data structure (such as a filehandle) then one option is to store the perl callback in a static hash associated with the data structure and then retrieve the relevant SV from the hash as required.
3. If the callback is passed arguments that do not identify the source (such as a text string from a file) then the only option is to write a number of callback functions in C and associate each one of them with a specific Perl callback. The disadvantage of this approach is that the number of callbacks is then limited to a fixed number.

* *This really does need an example but perlcall already covers this quite well*

The `Tk` module is an interesting example because it allows you to provide many more ways of specifying a callback than simply providing a code reference or subroutine name. Some examples are:

sub { }
\&somesub

A standard subroutine reference. This is used in Example 6-1

[\&somesub, $arg1, $arg2]

Invoke a subroutine with arguments. The arguments are grouped in an anonymous array.

["method", $object]

Invoke a method on an object. The arguments are grouped in an anonymous array.

The module achieves this flexibility by, in effect, storing the callback information in an object (in class `Tk::Callback`) and then invoking the Call method to run the callback itself.[16] If necessary, this approach can be used to simplify the C part of the callback code since the callback object can be created in perl (to contain whatever you want it to), and then passed to C and the C callback code can simply invoke the perl Call method on the object.

6.8. Other Languages

So far we have focussed on how to pass information between perl and C. This is because Perl itself is written in C and there are large numbers of libraries written in C. Since Perl is written in C it can therefore communicate with other languages that can be accessed from C. This section describes how to link Perl to two other languages. The first is C++ and the second is Fortran.

6.8.1. C++

C++ is an evolution of C (the name indicates that this language is a developed C language) with the addition of a true object-oriented framework. C++ compilers can compile C code as well as C++ code and when the OO framework is stripped away the core language is clearly based upon C.

In order to provide interfaces to C++ libraries you can use XS as before but this time in a C++ style. Since C compilers do not understand C++ and will not include the correct C++ infrastructure libraries we can no longer use the C compiler that was used to build perl. The first step in building a C++ interface is therefore to fix up the `Makefile.PL` file so that it uses your C++ compiler. This can be done simply by using the `CC` key to `WriteMakefile`:

```
use ExtUtils::MakeMaker;

WriteMakefile(
       NAME => 'CXXTest',
       VERSION_FROM => 'CXXTest',
       LD   => 'g++',
       CC   => 'g++',
);
```

The above program can be used to configure a module called `CXXTest` (we use `CXX` because we can't include ++ in a module name. A common alternative is to use `CPP`) and we replace the linker and the compiler command with **g++**. For compatibility with other operating systems you can either guess the compiler name at this stage (e.g. **CC** on Solaris; the approach we take later on with Fortran) or simply ask for it from the command line.

* *This section will probably cover a made-up simple class. I've recently been providing a perl interface to a C++ library covering a lot of the issues but that might be too specific. The biggest problem with C++ is that there can be multiple methods of the same name with differing calling signature so the XS code can be complicated trying to guess which c++ method to use. Also there is simply no documentation on how to use perl and C++ - have you read the section in perlxs?? It makes so many assumptions of knowledge.*
* *The real question is how much to talk about C++ itself? We spend two chapters describing C and here we are going on to object-oriented C in a couple of lines. Need to add the extern {} block around the C include files, need to mention that xsubpp automatically provides access to a THIS if it can guess one and guesses object constructors from the XS name. I think it is worth addign some of this but we have to be careful not to get bogged down in C++ specifics when we should be making the reader aware of C++ generalities.*

6.8.2. Fortran

Although Fortran has been around since the 60's with its popularity fading there are enormous numbers of scientific libraries available in this language and you might want to use them from Perl. Most Fortran implementations allow for the object code and libraries to be included in C programs but the following issues must be dealt with when doing so:

Pass by reference

All arguments are passed to Fortran subroutines as pointers and it is not possible to pass by value[17]

Types

The variable types in Fortran are similar (there are integers, floating point numbers and strings) but you must be aware of the differences in representation. For example, it is possible that the number of bytes used to represent an INTEGER in Fortran is different to the default int in C. Usually this is not an issue.

Strings

Fortran requires that the lengths of all strings are specified. Rather than using a null as a string terminator the compiler automatically passes a length argument to subroutines in addition to the pointer to the string itself. When passing strings between C and fortran additional arguments are required since the C compiler will not add them automatically. The position of these arguments depends on the compiler, the most popular approach in Unix systems is to simply add all the lengths in order to the end of the calling list (the approach taken with **g77**, Sun Fortran and Digital Unix Fortran) but it is also possible that the lengths might be added into the argument list immediately following the string in question (e.g. with Microsoft Visual C++ and Fortran). When sending a C string to a Fortran subroutine the string should be padded with blanks (if it is bigger than the current contents) and when returning a string from fortran the null should be added.

Packing order

Perhaps the largest difference between C and Fortran is the order with which multi-dimensional arrays are stored in memory. Fortran arranges arrays in column-major order (the column index is incremented before the row index in memory) whereas C arranges arrays in row-major order (the row index is incremented before the column index). Example 6-2 shows this difference when using a 3 by 2 array containing the numbers 1 through 6. The top diagram shows the order of the elements in memory, the middle diagram shows how C would arrange these elements and the lower shows how Fortran would arrange them. This means that element [1][1] in C will not be the same as element (2,2) in Fortran (additionally, C starts counting at zero whereas Fortran starts counting from 1). If you wish to pass a multi-dimensional array between C and Fortran it will have to be translated into the correct order.

Example 6-2. Array packing in C and Fortran

Comparison of row-major and column-major organisation of arrays

String arrays

Since strings in Fortran are of a specified length, arrays of strings in Fortran are simply contiguous blocks of memory rather than arrays of pointers. For strings of length 20, this is simply equivalent to the perl code **`$packed = pack("a20",@array);`**. This makes it very easy to pass string arrays from Perl to Fortran.

Linker names

Some compilers (especially on Unix) append an underscore (_) to the end of the subroutine name when the object code is generated. For example, a Fortran subroutine called `MYSUB` would be stored in the object file as `mysub_`. You need to know whether this happens on your platform. If you don't know, the easiest approach is to compile a Fortran program and examine the symbols[18]

Linking

When it comes to linking a Fortran library with a C main it will also be necessary to link with the Fortran runtime library. This library is included automatically when a fortran compiler is used for the link but must be specified explicitly when linking from a C compiler.

Now that we have described the issues involved in calling Fortran subroutine libraries from C we will now provide a quick example, no arrays, of how to do this from XS by providing some glue to talk to the PGPLOT Fortran library[19]. We will use the following simple subroutines from this library:

```
PGEND();


PGSCR(INTEGER CI, REAL CR, REAL CG, REAL CB);


INTEGER PGBEG(INTEGER UNIT, CHARACTER*(*) FILE, INTEGER NXSUB, INTEGER
NYSUB);


PGQINF(CHARACTER*(*) ITEM, CHARACTER*(*) VALUE, INTEGER LENGTH);
```

which on linux with **g77** have the following signature in C:

```
void pgend_();


void pgscr_(int *ci, float *cr, float *cg, float *cb);


int pgbeg_(int *unit, char *file, int *nxsub, int *nysub, int len_file);


void pgqinf_(char *item, char *value, int *length, int len_item, int
len_value);
```

Once this translation is known the XS code becomes quite straightforward:

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

#define F77BUFFER  256                                          ❶

#define pgend pgend_                                            ❷
#define pgscr pgscr_                                            ❷

MODULE = PGPLOT   PACKAGE = PGPLOT

void
pgend()                                                         ❸

void
pgscr(ci, cr, cg, cb)                                           ❹
  int &ci
  int &cr
  int &cg
  int &cb

int
pgbeg(unit, file, nxsub, nysub)                                 ❺
  int &unit
  char * file
  int &nxsub
  int &nysub
 CODE:
  RETVAL = pgbeg_(&unit, file, &nxsub, &nysub, strlen(file));❻
 OUTPUT:
  RETVAL

void
pgqinf(item, value, length)                                     ❼
  char * item
  char * value = NO_INIT
  int length = NO_INIT
 PREINIT:                                                       ❽
  char buff[F77BUFFER];
 CODE:
  value = buff;
  pgqinf_(item, value, &length, strlen(item), F77BUFFER);  ❾
  value[length+1] = '\0';                                       (10)
 OUTPUT:
  value
  length
```

❶ We use this definition to specify the maximum size of static buffers required to receive strings from Fortran subroutines.

❷ We can set up pre-processor symbols that will allow us to use the expected names in our XS code but will tell the C compiler to look for the Fortran name. In principal all the functions could be defined in this way. For this example, it is only convenient to do this for simple functions (those not using character strings, arrays or multiple return values) that can be dealt with automatically by the XS compiler without CODE blocks.

❸ `pgend` is simple because the XS compiler assumes that it can call a C function `pgend` even though the C pre-preprocessor will translate that text to `pgend_`. Without the definition at the top of the file the XS name will be different to the function name so a CODE block would be required.

❹ `pgscr` function takes 4 integer arguments but must pass them to the Fortran subroutine using pointers. The XS compiler does this automatically since the variables are declared with an ampersand as prefix.

❺ This function is not as simple. It takes 4 input arguments but one of those is a string.

❻ The integer arguments are passed in as pointers with the length of the string added as an extra argument to the C routine. We do not have to convert this string to Fortran format because we know the length in C and can simply inform the Fortran compiler of that length directly.

❼ This function has one input string and two output values. The NO_INIT tag is used to indicate that the input values for those variables are irrelevant.

❽ Allocate memory for a string to receive the output string.

❾ Since there are two string arguments, two lengths are appended to the argument list. The first is the length of the input string and the second is the length of the string output string. In principal the output string should be padded with blanks before sending it to Fortran (that step has been omitted for clarity).

(10) The string that comes back from Fortran will not be nul-terminated. Before sending it back to Perl the string must be made compatible with C. In this example we know the length of the string returned to us so we can simply append the terminator. In general we would need to step back through the string one character at a time until we found a non-blank character.

Once the XS code is written, the final issue is linking. All Fortran libraries require that the Fortran runtime library is included in the build. In order to simplify the task of keeping track of the runtime libraries required on different systems the `ExtUtils::F77` module can be used. This module attempts to determine the libraries that are required to link Fortran programs on your system with the relevant Fortran compiler. A `Makefile.PL` for the above XS code that utilises this module would look like this:

```
use ExtUtils::MakeMaker;
use ExtUtils::F77;                                          ❶

# See lib/ExtUtils/MakeMaker.pm for details of how to influence
# the contents of the Makefile that is written.
WriteMakefile(
    'NAME'              => 'PGPLOT',
    'VERSION_FROM'      => 'PGPLOT.pm', # finds $VERSION
    'PREREQ_PM'         => {}, # e.g., Module::Name => 1.1
    'LIBS'              => [ "-L/usr/local/lib -lpgplot " .
```

```
                              "-L/usr/X11R6/lib -lX11 " .   ❷
                              ExtUtils::F77->runtime ],     ❸
);
```

❶ Load the `ExtUtils::F77` library.

❷ The PGPLOT library requires X libraries for some devices

❸ The runtime libraries can be returned using the runtime class method.

Now when we run this program we get the following output (with **g77**):

```
Loaded ExtUtils::F77 version 1.14
Found compiler g77
ExtUtils::F77: Using system=Linux compiler=G77
Runtime: -L/usr/lib/gcc-lib/i386-redhat-linux/2.96 -L/usr/lib -lg2c
          -lm -L/usr/lib/gcc-lib/i386-redhat-linux/2.96 -lgcc
ExtUtils::F77: Validating -L/usr/lib/gcc-lib/i386-redhat-linux/2.96
          -L/usr/lib -lg2c -lm -L/usr/lib/gcc-lib/i386-redhat-linux/2.96
          -lgcc   [ok]
ExtUtils::F77: Compiler: g77
ExtUtils::F77: Cflags: -O
Writing Makefile for PGPLOT
```

Where `ExtUtils::F77` has determined that we will be using the **g77** compiler and that we need the `g2c` runtime library.

6.8.2.1. Interface Libraries and Portability

The issue of portability depends on the particular circumstances of your module. If you just want to use it on, for example, Solaris and Linux where the Fortran implementations are compatible you can simply write XS as described above. If on the other hand the intention is for your module to be adopted widely you will need to consider other operating systems and Fortran compilers. The `ExtUtils::F77` helps in this regard since it can work out the libraries for many different compiler and OS combinations. Additionally it can be used to determine whether a trailing underscore is required on function names (this ability can be used to set a C pre-processor macro).

There are a number of packages available to simplify the writing of portable interface code. One of the most extensive of these is the CNF package[20] written by Starlink (http://www.starlink.rl.ac.uk)[21]. CNF provides an large set of C pre-processor macros for dealing with variable types and calling conventions. There is also a support library that can be used to convert types (for example, strings and arrays) between the languages. Finally, CNF comes with very detailed documentation on the issues associated with mixed programming as well as the use of CNF itself. In order to demonstrate this alternative approach, here is the XS code for `pgqinf` re-written to use CNF:

```
void
```

```
fpgqinf(item, value, length)
  char * item
  char * value = NO_INIT
  int length = NO_INIT
 PREINIT:
  DECLARE_CHARACTER(fitem, F77BUFFER);
  DECLARE_CHARACTER(fvalue, F77BUFFER);
  DECLARE_INTEGER(flength);
  char buff[F77BUFFER];
 CODE:
  cnfExprt(item, fitem, fitem_length);
  F77_CALL(pgqinf) ( CHARACTER_ARG(fitem),
     CHARACTER_ARG(fvalue),
     INTEGER_ARG(&flength)
     TRAIL_ARG(fitem)
     TRAIL_ARG(fvalue) );
  length = (int)flength;
  value  = (char *)buff;
  cnfImprt( fvalue, length, value );
 OUTPUT:
  value
  length
```

Immediately obvious is the extra verbosity inherent in this approach and that in order to guarantee the correct types are being used for the Fortran layer a certain amount of copying is involved to go from the C to Fortran. This example also uses the translation functions `cnfExprt` and `cnfImprt` to export and import strings. Just to prove that CNF is doing the *right* thing, here is the same code block with the macros expanded (using **g77**):

```
char fitem[256]; const int fitem_length = 256;
char fvalue[256]; const int fvalue_length = 256;
int flength;
char buff[256];
cnfExprt(item, fitem, fitem_length);
pgqinf_ ( fitem,
          fvalue,
          &flength
          ,fitem_length
          ,fvalue_length );
length = (int)flength;
value = (char *)buff;
cnfImprt( fvalue, length, value );
```

It is clear that the trade-off for portability in this case is counter balanced by a decrease in speed of the resulting code. For maximum performance it is probably best to make some assumptions.

6.8.2.2. Interface considerations

The interface provided in this Fortran example is not very *perl-y*. As described in Section 2.6 the needs of a particular library should be addressed on a case-by-case basis. The PGPLOT library has a well defined API that has been in use for many years and there are some benefits to following that interface. When migrating from C or Fortran applications it might be easier, both for the author of the module (less documentation to write, no complications with PPCODE blocks) and for the user (no need to learn a new interface) if the Perl port of the library looks as close as possible to that implemented in other languages. This is the approach taken with the `PGPLOT` module on CPAN. For the routines described above, the most likely candidate for change is `pgqinf`. This routine has one input argument and two return arguments. One of these arguments is simply the length of the string and is not required by Perl. A much better approach from Perl's viewpoint is probably:

```
$value = pgqinf($item);
```

The Perl Data Language goes a step further. The PGPLOT interface in PDL is completely divorced from the underlying library API and uses an object-oriented layer to implement plotting. This has made it easier for users of PDL since different plotting engines can be used with only minimal changes to the perl code.

6.8.3. Java?

* *There is of course JPL (Java Perl Lingo). From the XS point of view I'm only interested in the JNI module (along with the base JPL module) which allows for java code to be called from Perl. The bulk of JNI exists for embedding a perl interpreter inside java rather than invoking java methods from perl. JNI does all the hard work so that the author of the Java code does not actually use any XS directly so it doesn't really sit well in this chapter even if we wanted to talk about it. I have tried to build JNI (on linux with JDK1.1.7 and 1.3.0) and I have not had much success so far - builds are okay but it refuses to launch a new JVM. I have not tried it on Solaris with an old JDK). There does not seem to be that much going on with JPL in general so it might be safer simply to ignore it. Alternatively, Inline::Java seems to come with a JNI interface and does seem to work with modern JDKs (that built and tested with no problems on linux).*

6.9. Interface Design - Part 2

In Section 2.6 we covered the basic interface issues that should be thought about before writing even the simplest XS module. Now that we have a more thorough grounding in XS we will summarise the design issues that have been brought up during this chapter.

- Look at the functions in the external library. Is the C library using objects in disguise? If it is repeatedly using an internal opaque structure then it is probably best to use that structure as an object in Perl.

- Multiple return arguments should be returned as proper return arguments rather than as input arguments that are modified. Use PPCODE to adjust the return stack to suit.
- If a status variable can only have two states consider translating the C value to a Perl true or false rather than matching the interface provided by the external calling convention. Alternatively, if the function only returns useful information when the status is good, do not return the status value at all. Return the values if everything is okay, and an empty list or `undef` otherwise. The `XSRETURN_*` functions are provided for this purpose.

  Using our example from Section 2.6 we can then change a signature of:

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

  to

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

  using the following XS code:

  ```
  double
  compute( factor )
    int factor
   PREINIT:
    int status;
   CODE:
    status = compute( factor, &RETVAL);
   OUTPUT:
    RETVAL
   CLEANUP:
     if (status == -1)
       XSRETURN_UNDEF;
  ```

- C++ context-based method overloading provides a particular headache for interface design. If it is possible to determine the correct method from the argument stack (for example by counting the number of arguments) then it is possible to have a single perl method that calls the required C++ method. If the distinction is simply by type then it may be necessary to either have distinct perl methods (one for each type) or a single perl method that is uses a relevant perl type (since perl does not care about the difference between an integer, double or string) . This issue is disccussed in Section 6.8.1

Finally, don’t be scared of using perl. Many of these examples can be simplified by using a perl intermediary layer (for example, using `pack`/`unpack` to handle arrays). If you feel stronger in perl than in C this is a valid approach. If you feel that you need the extra speed you can recode it in C without

needing to change the external interface to the module. The important thing is to have a clean public interface.

6.10. What's Really Going On?

So far in this book, we have created XS files, typed **make** and watched whilst lots of interesting things scroll off the screen. In this section we will take a closer look at what is really happening to these XS files and how XSUB really interface to the Perl internals.

The XS interface exists to simplify the way user-supplied C code can be called from within Perl by being one layer above the perl internals. In principal it is possible to write the C code directly without going through the XS layer at all but this approach is not recommended since (a) it is more complex than is required for simple interfacing, (b) it is repetitive and error prone when providing an interface to many external functions and (c) may suffer compatibility problems if the internals are changed.

6.10.1. What does xsubpp generate?

By now we know that the first step in creating an XS module is for the **xsubpp** command to process the XS file in conjunction with a typemap file and from these generate a C file that can be compiled. Here we will revisit some of the XS examples from Section 2.4 but this time we will take a look at the `.c` output files. To refresh our memories here is the XS file that we will be experimenting with:

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

#include <stdio.h>
#include <string.h>


size_t strconcat (char* str1, char* str2, char* outstr) {
   strcpy( outstr, (const char*)str1 );
   strcat( outstr, (const char*)str2 );
   return strlen( outstr );
}

MODULE = Example PACKAGE = Example

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

```
size_t
strconcat( str1, str2, outstr )
  char* str1
  char* str2
  char* outstr = NO_INIT
 PREINIT:
  size_t length;
 CODE:
  length = strlen( str1 ) + strlen( str2 ) + 1;
  New(0, outstr, length, char );
  RETVAL = strconcat( str1, str2, outstr );
 OUTPUT:
  outstr
  RETVAL
 CLEANUP:
  Safefree( outstr );
```

If we now build this module we will find a C file called `Example.c` in the working directory. This file will contain the something like the following:

* *This is crying out for extensive annotation but I don't feel that callouts can really do what I want to do here. Should be a "hedgehog" diagram but I'm loathe to try and do one in xfig. Need to investigate drawing packages.*

```
/*
 * This file was generated automatically by xsubpp version 1.9507 from the
 * contents of Example.xs. Do not edit this file, edit Example.xs instead.
 *
 * ANY CHANGES MADE HERE WILL BE LOST!
 *
 */

#line 1 "Example.xs"
#include "EXTERN.h"                                              ❶
#include "perl.h"
#include "XSUB.h"

#include <stdio.h>
#include <string.h>


size_t strconcat (char* str1, char* str2, char* outstr) {
   strcpy( outstr, (const char*)str1 );
   strcat( outstr, (const char*)str2 );
   return strlen( outstr );
}

#line 25 "Example.c"
XS(XS_Example_treble)
{
    dXSARGS;
    if (items != 1)
Perl_croak(aTHX_ "Usage: Example::treble(x)");
```

```
    {
int x = (int)SvIV(ST(0));
int RETVAL;
dXSTARG;
#line 21 "Example.xs"
   RETVAL = 3*x;
#line 37 "Example.c"
XSprePUSH; PUSHi((IV)RETVAL);
    }
    XSRETURN(1);
}

XS(XS_Example_strconcat)
{
    dXSARGS;
    if (items != 3)
Perl_croak(aTHX_ "Usage: Example::strconcat(str1, str2, outstr)");
    {
char* str1 = (char *)SvPV(ST(0),PL_na);
char* str2 = (char *)SvPV(ST(1),PL_na);
char* outstr;
#line 31 "Example.xs"
  size_t length;
#line 54 "Example.c"
size_t RETVAL;
dXSTARG;
#line 33 "Example.xs"
  length = strlen( str1 ) + strlen( str2 ) + 1;
  New(0, outstr, length, char );
  RETVAL = strconcat( str1, str2, outstr );
#line 61 "Example.c"
sv_setpv((SV*)ST(2), outstr);
SvSETMAGIC(ST(2));
XSprePUSH; PUSHi((IV)RETVAL);
#line 40 "Example.xs"
  Safefree( outstr );
#line 67 "Example.c"
    }
    XSRETURN(1);
}

#ifdef __cplusplus
extern "C"
#endif
XS(boot_Example)
{
    dXSARGS;
    char* file = __FILE__;

    XS_VERSION_BOOTCHECK ;

        newXS("Example::treble", XS_Example_treble, file);
        newXS("Example::strconcat", XS_Example_strconcat, file);
```

```
    XSRETURN_YES;
}
```

The global things to get from this file are the following:

- The C code that appears in the XS file before the MODULE keyword is passed through unchanged.
- Each XS function appears as a real C function but with a modified name. The name is derived by concatenating the fixed prefix `XS_`, a C-ified form of the current PACKAGE (as defined by the PACKAGE keyword in the XS file) and the function name itself. If a PREFIX is defined then that is removed from the function name before concatenation. For example, if the function name is `slaMap`, with a prefix of "sla" and a package name of `Astro::SLA`, the internal function name will be `XS_Astro__SLA_Map`. Here the colons have been replaced with underscores.
- If there are a fixed number of arguments, code has been inserted to check the correct number are on the stack and to issue an error message if necessary.
- Whenever there is a shift from original code to auto-generated code, the line numbering is changed to indicate the source of the code. This is achieved using the `#line` special directive[22]. This means that if an error occurs in an XS line any compiler error messages will point to the correct line and file name rather than to the line in the translated C code. This simplifies debugging enormously.
- The final section is completely auto-generated. The boot function is responsible for registering the XS functions with perl and is called automatically by DynaLoader when the module is loaded.

Now that we have a global overview of the file contents, we can cover the relationship between the XS definition of a function and the final C code. The XS sections are propogated to the C translation without changing the inherent order. Arguments are declared first, then comes additional variable definitions (PREINIT), CODE blocks, the handling of output variables and finally, CLEANUP code is included to tidy things up. Of course there are many other types of XS declarations and they are included by **xsubpp** in the same way. Most of these steps simply entail the addition of some standard stack handling code or the insertion verbatim of the code supplied by the module author. The main complications involve variable input and output. These involve the stack handling and variable declarations. Let's take a closer look at the `strconcat` function shown above. The XS declaration and corresponding C code were:

```
char* str1
char* str2
char* outstr = NO_INIT
```

and

```
char* str1 = (char *)SvPV(ST(0),PL_na);
char* str2 = (char *)SvPV(ST(1),PL_na);
char* outstr;
```

xsubpp does the following to generate this code:

1. Look up char* in the typemap file(s).

2. Look up T_PV in the INPUT section.
3. See if the INPUT entry matches /\t\$var =/.
   - If it matches then the typemap entry is assumed to be a single statement and the variable can be declared and assigned at the same time. This is the case with T_PV which has a typemap entry of **`<tab>$var = ($type)SvPV($arg,PL_na)`**.
   - If it does not match this (for example, because the variable is to be type-checked before copying as is the case with T_AVREF) the variable is simply declared and the typemap code is deferred until after the other variable declarations (after the remaining arguments are declared and after PREINIT blocks). This is necessary since C does not allow varaiables to be declared midway through a code block.
4. Replace all the Perl variables in the typemap entry with the corresponding values and print them to the C file. Recall that `ST(n)` refers to an SV on the argument stack (Section 6.4). In this example positions 0 and 1 are relevant and nothing is copied from position 2.

Return variables are dealt with in a similar way unless a PPCODE block is used. In our `strconcat` function the output code for `outstr` is simply the typemap entry with variable substitutions. The `RETVAL` return variable is dealt with in the same manner except **xsubpp** recognizes the standard variable types and translates them into the corresponding PUSH commands rather than using the OUTPUT typemap entry.

6.11. Summary

In this chapter we have extended our knowledge of XS to cover files, hashes, arrays and callbacks and have seen how the XS code and typemap files translate into code suitable for use by the Perl internals. We have also seen how write XS code suitable for interfacing perl to libraries written in Fortran and C++.

In subsequent chapters we will step further away from the internals and see what systems are available to simplify the linking of perl to external libraries.

6.12. Further Reading

* *It would be really nice to provide some kind of appendix explaining the different "standard" typemaps. Some of them are very strange and all are essentially undocumented. For now, simply reference my XS::Typemap documentation (part of perl 5.7.1) where I have documented most of them.*

perlxstut manpage

This XS tutorial comes with Perl and provides several XS examples.

perlxs manpage

: This manpage contains detailed information on XS.

perlcall manpage

: This manpage from the standard distribution contains a lot of information on setting up callbacks.

XS::Typemap

: Starting with Perl 5.7.1, the source distribution includes the `XS::Typemap` module that is used to test that the typemap entries behave as expected. This module is not installed but the source contains examples for most of the typemap entries.

Astro::SLA
PGPLOT

: The `Astro::SLA` and `PGPLOT` modules on CPAN both contain helper code that can be used to pack and unpack arrays with the minimum of effort. The code in the `arrays.c` file from these distributions is useful for providing a pre-packaged solution for dealing with arrays without going to the expense of using PDL.

Notes

1. Although why you would want to use this function rather than the standard `strcat` function or `sv_catpv` from the Perl API is a mystery!
2. The alternative is to pre-allocate the memory inside Perl by passing a string in as the third argument (and removing the NO_INIT declaration). The contents of the string will then be overwritten by `strconcat`. This will work but can generate core dumps if the string is not large enough to receive the result string. Not recommended!
3. The `Term::ReadLine::Gnu` module has a full implementation
4. or even **`IO::Handle->new_from_fd()`**
5. The location of the file on your system can be determined using Perl's `Config` module:

```
% perl -MConfig -MFile::Spec -e 'print File::Spec->catfile($Config{installprivlib},"ExtUtil
/usr/lib/perl5/5.6.0/ExtUtils/typemap
```

6. see `ext/POSIX/POSIX.xs` in the Perl source tree for details of the implementation of `POSIX::strftime`.
7. `gmtime` is implemented in Perl in the file `pp_sys.c`.
8. If you are happy to have a non-OO style constructor, simply changing the typemap entry would be enough. The calling style would then remain

```
$object = Time::gmtime();
```

   except that this would return an object.

9. Ignoring any preference in syntax of **`Time::timegm($tm)`** versus **`$tm->timegm`**
10. only if using an intermediary wrapper function. Do not ask people to provide information that perl already knows!

11. The primary developers of PDL were Karl Glazebrook (an astronomer), Tuomas Lukka and Christian Soeller

* *Need to check what Christian and Tuomas do*

12. Karl Glazebrook is English!
13. Version 1 of PDL stored the data as a packed string in an SV exactly as shown Section 6.6.1.1.3. Version 2 made the object more opaque.
14. Assuming your perl can support dynamic loading of libraries
15. Of course we can just use the PDL sum method directly!
16. The Call method in `Tk::Callback` is the perl interface to this but the methods are coded in C for efficiency. See the file `Event/pTkCallback.c` in the `Tk` source distribution for details.
17. Modern Fortran implementations provide the `%VAL` function to allow pointers to be passed by value from integers.
18. Use the **nm** command on unix systems.
19. The PGPLOT library is available from Caltech (http://www.astro.caltech.edu/~tjp/pgplot/) and can be used for generating scientific plots. The `PGPLOT` Perl module by Karl Glazebrook, using the C binding to this library, is available from CPAN (http://www.cpan.org)
20. The CNF package currently uses the Starlink Software Licence. This licence allows for redistribution of source code but currently restricts the library to non-commercial use. Moves are currently underway to open up the licence.
21. Starlink is a research organization in the United Kingdom. It provides data processing and analyis software for research astronomers in the UK.
22. Identical to the `#line` special comment supported by perl and described in the `perlsyn` manpage.

Chapter 7. Alternatives to XS

So far we have done all of our Perl interfaces to C using XS and have demonstrated just how powerful and complex XS can be. In this chapter we are going to take a step above XS and show how it is possible to create an interface to C without using XS or reading page after page of documentation.

We will begin by looking at one of the earliest alternatives to XS and then proceed to the current front-runner for XS simplification. On the way we will address the special demands called for by the Perl Data Language and finish up by covering some of the less useful approaches to ignoring XS. We will make use of the examples from previous chapters to demonstrate the differences and similarities of these approaches.

Nearly *all* the schemes described in this chapter are implemented as an abstraction layer above XS itself either by generating `.xs` files to be passed to **xsubpp** or generating the code equivalent to the output of **xsubpp** that interfaces directly to perl's internal XS system. This approach has the key advantage that a change to the XS system will only require a fix to the software that processes these definition files, all code that uses these systems will not require modification.

7.1. h2xs

By far the oldest attempt at providing a simplified access to XS is the **h2xs** program that comes with Perl. We have already used this command in Chapter 2 to generate a basic framework for perl and XS-based modules but it can be used to automatically generate XS wrapper code for simple libraries. As the name of the program suggests, it can take a C header file as input and generate a `.xs` file as output. We will not spend much time describing this program in detail but as an example let's see what happens when we run this command on a header file containing prototypes for the `print_hello` and `treble` examples used in Section 2.3. We can call this file `example.h`:

```
void print_hello ();
int treble (int x);
```

You can then run **h2xs** with the `-x` option to force it to scan the header file looking for function prototypes (for this to work you will need to download the `C::Scan` module from CPAN). Without this option **h2xs** will still attempt to read the include file but it will only be able to extract #define's rather than full XS entries. For our example we get the following output:

```
% h2xs -x -c -A example.h
Scanning typemaps...
 Scanning /usr/lib/perl5/5.7.2/ExtUtils/typemap
Scanning example.h for functions...
Scanning example.h for typedefs...
Writing Example/Example.pm
Writing Example/Example.xs
Writing Example/Makefile.PL
```

```
Writing Example/README
Writing Example/t/1.t
Writing Example/Changes
Writing Example/MANIFEST
```

The additional options turn off the code for handling constants since this module won't be using any. The output file `Example.xs` contains the following:

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

#include <example.h>


MODULE = Example PACKAGE = Example


void
print_hello()

int
treble(x)
int x
```

and looks identical to that used in Section 2.3 except that the actual code for the functions is missing (you would have to put that after the include directives or else create a library to link against).

In general **h2xs** is fine for handling simple interfaces using scalar arguments but rapidly runs out of steam. Even with a seemingly simple function such as `strconcat` from Section 6.1 **h2xs** gets the wrong answer:

```
STRLEN
strconcat(str1, str2, outstr)
        char *  str1
        char *  str2
        char *  outstr
```

It assumes that all arguments are for input and, additionally, the C interface requires that memory is allocated for the output string before use; something that **h2xs** can not possibly guess.

If you intend to use **h2xs** for anything but the simplest interface be prepared to do a lot of work on the file that is automatically generated for you.

7.2. SWIG

SWIG (standing for Simplified Wrapper and Interface Generator) was developed in 1995 by David Beazley at Los Alamos to simplify the writing of interface code between scripting languages and C/C++ libraries. First for a little known language, and then, in 1996, to cover Tcl, Perl and Guile. Since then it has been developed to cover many more languages including Java. One of the major advantages of SWIG over XS is that it allows you to write a single interface specification and use it for all the languages supported by SWIG.

Let's start our investigation of SWIG by seeing how to provide a perl interface to the `sinh` math function (this already exists in the `POSIX` module) and a related constant. All SWIG interfaces begin with the definition of the generic interface definition. For `sinh` this would look something like:

```
%module Simple                                  ❶

double sinh ( double ang );                     ❷
#define PI 3.141592654                          ❸
```

❶ All SWIG files begin by declaring the module name (if it is not defined in here it must be specified on the command line). SWIG commands always start with a per cent sign.

❷ This is simply the C prototype for the function we are trying to call. SWIG can be used to simply parse a C include file and generate a language interface directly. In general this is not the recommended approach since C include files tend to include many functions and data structures that are not required for the interface itself.

❸ A C pre-processor define will be treated as a constant variable. Constants can also be defined using the `%constant` command.

If we now save the above description to a file `simple.swg` (`simple.i` is a common alternative) we can run SWIG on this file and generate the code required to interface the target language to the library function:

```
% swig -perl5 simple.swg
```

Here we use the `perl5` option to generate code for Perl rather than another language. Once executed, this command writes a file `simple_wrap.c` containing all the code necessary to call our function from Perl and a small perl wrapper module called `Simple.pm`. With version 1.3.9 of SWIG[1] this C output file is approximately 10kB; impressively large for a 2 line interface description file. The important thing to realise is that the output code is C that can be compiled and linked as a module directly as **xsubpp** is not required here. To convince yourself this is the case look in `simple_wrap.c` and you will find the following XS code (or something similar) for `sinh`:

```
XS(_wrap_sinh) {
    double arg0 ;
```

```
    int argvi = 0;
    double result ;
    dXSARGS;

    if ((items < 1) || (items > 1))
    croak("Usage: sinh(ang);");
    arg0 = (double )SvNV(ST(0));
    result = (double )sinh(arg0);
    ST(argvi) = sv_newmortal();
    sv_setnv(ST(argvi++), (double) result);
    XSRETURN(argvi);
}
```

Unsurprisingly this is very similar to the code generated by **xsubpp** (see for example Section 6.10). Once you've got this C code the next step is to compile it and link it as a shared library such that it can be used as a module. Rather than doing this manually we can do it using `MakeMaker` as for any other module. Here is the `Makefile.PL`:

```
use ExtUtils::MakeMaker;

WriteMakefile(
              'NAME'      => 'Simple',
              'VERSION'   => '0.01',
              'OBJECT'    => 'simple_wrap.o',
);
```

The key difference here between an XS module and a SWIG module is that we don't need to run **xsubpp** and `MakeMaker` just needs to compile a C file. We do this by using the `OBJECT` hash key to tell `MakeMaker` the names of the object files that we wish to use to form the module. The makefile will automatically add the code for compiling the corresponding C files to object files. We can now build the module as we would any other XS module:

```
% perl Makefile.PL
Writing Makefile for Simple
% make
cp Simple.pm blib/lib/Simple.pm
cc -c   -fno-strict-aliasing -I/usr/local/include -D_LARGEFILE_SOURCE -D_FILE_OFFSET_BITS=6
O2   -DVERSION=\"0.01\" -DXS_VERSION=\"0.01\" -fpic -I/home/timj/test/lib/perl5/5.7.2/i686-
linux/CORE   simple_wrap.c
Running Mkbootstrap for Simple ()
chmod 644 Simple.bs
rm -f blib/arch/auto/Simple/Simple.so
LD_RUN_PATH="" cc  -shared -L/usr/local/lib simple_wrap.o
    -o blib/arch/auto/Simple/Simple.so
chmod 755 blib/arch/auto/Simple/Simple.so
cp Simple.bs blib/arch/auto/Simple/Simple.bs
chmod 644 blib/arch/auto/Simple/Simple.bs
```

You could include a makefile dependency for the SWIG input file itself (running SWIG on it when it is modified) but for a distribution that is to be put distributed you probably will not want to add a SWIG dependency to your distribution). We can now run it to see if it does what we expect:

```
% perl -Mblib -MSimple -e 'print Simple::sinh(3)'
10.0178749274099
% perl -Mblib -MSimple -e 'print $Simple::PI'
3.141592654
```

As an ever so slightly more complicated example let's take a look at how we would write an interface file for the `treble` of Section 2.3:

```
%module Treble

%{

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

%}

int treble (int x);
```

The difference here is that C code has been inserted into the definition. The code between `%{` and `%}` is copied directly from the definition file to the output C code. This is where you can put include file loading as well as actual C code. This module can be built in exactly the same way as the `Simple` module.

7.2.1. Arrays and Structs

As already mentioned, one thing that distinguishes SWIG from other approaches such as XS or `Inline` is its language agnostic approach. In many cases this is beneficial but when it comes to dealing with complicated data structures the approach usually leads to modules that do not fit in well with the language philosophy. In SWIG all complex data types (pointers) are treated as opaque objects. Arrays and structures are always treated as pointers. In most cases this is fine since many libraries use structures as opaque entities already and simply pass the pointer from function to function. If the structure is defined SWIG will generate accessor methods to enable access to the individual structure components.

* *I think this needs to be rewritten - I don't like the structure (no pun intended) [TJ]*

In this section we'll examine the use of structures from SWIG by looking at the `gmtime` and related functions that we used for the examples in Section 6.5. Here is a simple SWIG interface to `gmtime`, `asctime` and `mktime`:

```
%module Time
```

```
%{
#include "time.h"
%}

typedef int time_t;

%typemap(perl5, in) time_t * (time_t temp) {
  temp = SvIV( $source );
  $target =
}

struct tm *gmtime(time_t * timep);
char * asctime( const struct tm *timeptr);
time_t mktime( struct tm *timeptr);
```

This looks really straightforward except for the complication of the typemap definition (the `typedef` is there simply to tell SWIG that a `time_t` can be treated as an integer). The problem is that `gmtime` expects a pointer to a variable. We could write a wrapper C routine that allocates memory and populates the value, returning a pointer to perl that can be passed to this function but it is much simpler from the perl perspective to write a typemap that will convert a simple integer (usually the output of the perl `time` function) and convert it to a pointer. If we build this module and run we can see the following:

```
% perl -Mblib -MTime -e 'print Time::gmtime(time)'
_p_tm=SCALAR(0x8107cc8)
% perl -Mblib -MTime -e '$t=Time::gmtime(time); print Time::asctime($t)'
Tue Nov 13 02:22:55 2001
```

SWIG has provided us with an interface even though it doesn't know what the contents of a `tm` structure are. All you get is a blessed scalar of an unhelpful class (in this case `_p_tm` since this is a pointer to a `tm`) which SWIG can use for internal type consistency checking. If you want to be able to look inside the structure you simply tell SWIG what it contains:

```
%module Time

%{
#include "time.h"
%}

typedef int time_t;

%typemap(perl5, in) time_t * (time_t temp) {
  temp = SvIV( $source );
  $target =
}

struct tm
{
  int tm_sec;                   /* Seconds.     [0-60] (1 leap second) */
```

```
  int tm_min;                   /* Minutes.     [0-59] */
  int tm_hour;                  /* Hours.       [0-23] */
  int tm_mday;                  /* Day.         [1-31] */
  int tm_mon;                   /* Month.       [0-11] */
  int tm_year;                  /* Year - 1900.  */
  int tm_wday;                  /* Day of week. [0-6] */
  int tm_yday;                  /* Days in year.[0-365] */
  int tm_isdst;                 /* DST.         [-1/0/1]*/
};


struct tm *gmtime(time_t * timep);
char * asctime( const struct tm *timeptr);
time_t mktime( struct tm *timeptr);
```

Now if we use this module we can find out the time from the structure:

```
% perl -Mblib -MTime -e '$t=Time::gmtime(time); print Time::tm_tm_year_get($t)'
101
```

SWIG automatically creates accessor function (there is a corresponding set method). In this case the function is an accessor for the `tm_year` field in the struct `tm` so the name is a bit repetitive. Since this approach is clearly not going to win any fans with users of your class SWIG provides an option for handling structures as true Perl objects. If you use the `-shadow` option to SWIG the interface looks much more agreeable:

```
% swig -perl5 -shadow time.swg
% make
...
% perl -Mblib -MTime -e '$t=Time::gmtime(time); print $t'
Time::tm=HASH(0x8144648)
% perl -Mblib -MTime -e '$t=Time::gmtime(time); print $t->mktime;'
1005656269
% perl -Mblib -MTime -e '$t=Time::gmtime(time); print $t->{tm_year}'
101
```

The above example shows that the structure is converted to an object in class `Modulename::structname` and can be used to invoke methods (in this case `mktime`) and to access object instance data via a tied hash interface.

* *Still to do with SWIG section: Something on memory management, an explicit example of array use (including N-Dim arrays with accessor methods), maybe more explicit typemap examples. Again, depends on how deeply we want to go before moving people onto the SWIG documentation. 4.5 pages so far so can probably justify a couple more to round the chapter out at approximately 20 pages (without diagrams).*

7.3. The `Inline` module

The `Inline` modules were first developed in 2000 by Brian Ingerson in order to simplify the XS learning curve. The family of `Inline` modules allow you to write non-Perl code within the body of your Perl program with the expectation that Perl will know what to do with it. This can be demonstrated by showing the `Inline` version of our first XS examples from Section 2.3.

> `Inline` is not distributed as part of Perl. You will need to download it and its dependencies (such as the `Parse::RecDescent` module) from CPAN before you can run any of these examples

```
use Inline "C";

print treble(5),"\n";
&print_hello;

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

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

If you run this program it does exactly what you expect. No need to create a module and work out the XS syntax; it just works. You should get the following output:

```
15
hello,world
```

The first time you run it it will probably take a few seconds to execute. The second time it will run as fast as you would expect the XS version to run.

`Inline` itself is not limited to C. Modules exist for writing and calling C++, Python and Java code from perl. As an introduction to the techniques of using `Inline` we will keep our description to the C implementation but if you are interested in other languages CPAN will probably have them all covered!

7.3.1. What is going on?

Before we rush headlong into more examples, it may be instructive to explain what is really happening here.[2] When you write a Perl program using `Inline` unsurprisingly a lot of work is going on behind the scenes. In outline, the following is occurring:

1. The module reads the code from the appropriate place (usually below the `DATA` handle).
2. An MD5 checksum is then calculated for the code in question.
3. This checksum and associated information (such as operating system and perl version number) is compared with any modules that may have been automatically generated on previous occasions (this all happens in the `_Inline` or `$HOME/.inline` directory unless specified otherwise). If it does not match with any existing checksum or configuration the code is analysed to determine the function names and calling arguments (maybe using a parser module such as `Parse::RecDescent`).
4. An XS module is then generated based on the functions and arguments in the inlined code and with a name derived from the checksum (If no name is specified the module will be the name of the file containing the original code with punctuation removed and the first few letters of the checksum).
5. The module is now built and installed into a local directory.
6. Alternatively, if the checksum matches with a checksum calculated during a previous run of the program the module is loaded directly with out being compiled.

In essence this is fairly straightforward. A module is built automatically if required, else it is simply loaded as for any other module. This explains why an inlined program takes a while to run the first time through but is then almost as fast as for a normal module on subsequent runs. The main complication is the parsing of the C code to generate XS specifcation.

7.3.2. Some more examples

Now that we have seen the magic and have an idea of what is really happening we should now tak a look at some more serious examples. Anything that simply passes some scalar arguments in and gets a return value should look just like a normal C function. In this section we'll cover memory allocation issues, lists and multiple return arguments since these are the most likely sources of confusion.

7.3.2.1. Strings and things

If your function returns a string (essentially a memory buffer) that has been created in your function then you will likely have to worry about memory allocation. We can examine this issue by using the string concatenation example of Section 6.1. This function took two strings and returned the concatenated string. In that example we used `New` and `Safefree` for the memory allocation. This was okay but it relied on the `CLEANUP` section of the XS code running after the memory buffer had been copied back onto the argument stack. With `Inline` we can not get away with this since we are being called by the XS routine so if we return a char* we have no way of freeing it after it has been returned. To overcome this problem we have two choices:

- Return an SV containing the result rather than returning a char*. This allows you to free the memory before returning from your function since the string is copied into an SV:

```
SV * strconcat( char * str1, char * str2 ) {              ❶
  char * outstr;
  SV* outsv;
  int len = strlen(str1) + strlen(str2) +1;
  New(0, outstr, len, char);                              ❷

  strcpy( outstr, (const char*)str1 );                    ❸
  strcat( outstr, (const char*)str2 );                    ❸

  outsv = newSVpv(outstr, len);                           ❹
  Safefree(outstr);                                       ❺
  return outsv;                                           ❻
}
```

❶ We mark this function as returning an SV*.

❷ Allocate a string buffer of the right size.

❸ These two lines are the core of the `strconcat` function described in earlier chapters. They represent a real-world call to an external library. Of course, implementing this function using the perl SV API is much easier!

❹ Create a new SV and copy the contents of the string buffer into it.

❺ Free the buffer since we don't need it any longer.

❻ Return the SV*. It is automatically marked as mortal by XS before being put onto the return stack.

This is a little inefficient since you actually end up allocating two string buffers: once for the string buffer you are using and once when you create the SV. In fact, since you are going to the trouble of using an SV you may as well use the SV's buffer directly:

```
SV * strconcat( char * str1, char * str2 ) {
  SV* outsv;
  int len = strlen(str1) + strlen(str2) +1;
  outsv = NEWSV(0, len);                                  ❶

  strcpy( SvPVX(outstr), (const char*)str1 );             ❷
  strcat( SvPVX(outstr), (const char*)str2 );             ❷

  SvPOK_on(outsv);                                        ❸
  SvCUR_set(outsv, len);                                  ❸
  return outsv;
}
```

❶ Create a new SV with a string buffer of the correct size.

❷ We use the `SvPVX` macro to obtain the char* pointing to the start of the buffer.

❸ Now that we have populated the string we need to tell Perl that the string component is valid and what its useful length is.

This has the added inconvenience of having to mark the SV as a valid string (`SvPOK`) and the length of it (`SvCur`) but you only allocate one buffer.

- Do what XS does and simply create a new SV with the result and push it onto the stack ourselves. This is essentially identically to returning an SV* to `Inline` but cuts out the middle man. We'll describe how to manipulate the argument stack from `Inline` in the next section.
- If you really need to allocate a buffer and return it as a non-SV then you will have to use a memory allocator that makes use of mortal SVs:

```
char * strconcat3( char * str1, char * str2 ) {
  char* outstr;
  int len = strlen(str1) + strlen(str2) + 1;
  outstr = (char *)get_mortalspace( len );                ❶
  strcpy( outstr, (const char*)str1 );
  strcat( outstr, (const char*)str2 );

  return outstr;
}

void * get_mortalspace ( size_t nbytes ) {                ❷
  SV * mortal;
  mortal = sv_2mortal( NEWSV(0, nbytes ) );
  return (void *)SvPVX(mortal);
}
```

❶ Allocate some memory for the buffer.

❷ This is the memory allocator described in Section 6.6.2. It is here explicitly rather than putting it in an external file. This does mean that `Inline` will create a perl interface for the function even though it is not required (if you call it, the function will allocate the requested number of points and free it before you get the return value. The return value will be a pointer).

The XS code will receive the char* put it into an SV and push it onto the argument stack. Once it is read from the stack the memory associated with the SV will be freed automatically.

7.3.2.2. Summing an array

In Section 6.6.1 we covered in detail how to handle perl arrays in XS in the form of a list, a reference and a packed string. Here we will show how to use `Inline` to deal with lists and an array reference.

First a reference. Since a reference is a scalar and a corresponding typemap entry exists this is fairly easy to do with `Inline`:

```
use Inline C;

print sum_as_ref([1..10]);

__END__
__C__

int sum_as_ref(AV* avref)
{
  int len;
  int i;
  int sum = 0;
  SV ** elem;

  len = av_len(avref) + 1;

  for (i=0; i<len; i++) {
    elem = av_fetch(avref, i, 0);
    if (elem != NULL)
       sum += SvIV( *elem );
  }
  return sum;
}
```

The above code is almost identical to the example using XS (for clarity we are doing the sum in place rather than calling the external C routine) and there is nothing new in it at all.

When processing lists things get more interesting. `Inline` generates the XS code for our function and then calls our function. This means that we can no longer rely on XS to provide all the stack handling facilities that we are familiar with. `Inline` overcomes this problem by providing some simple macros of its own for initialising the stack variables and manipulating it. These can be demonstrated in the following code (without the perl code removed since it will be nearly identical to the previous example):

```
int sum_as_list( SV* arg1, ... )                    ❶
{
  int i;
  int len;
  int sum = 0;
  SV* elem;
  Inline_Stack_Vars;                                ❷

  len = Inline_Stack_Items;                         ❸
  for (i=0; i<len; i++) {
    elem = Inline_Stack_Item(i);                    ❹
```

```
    sum += SvIV( elem );                                    ❺
  }
  return sum;
}
```

❶ Ellipsis (...) are used to indicate to `Inline` that multiple arguments will be used. At least one argument has to be declared even though you may be retrieving all the stack variables using the macros. In this case `arg1` is declared but is not used in the function directly (it is used via `Inline_Stack_Item(0)`).

❷ Initialise the stack-related variables used by the othe stack macros. This must always be placed in the variable declaration section of the function.

❸ This counts the number of arguments on the stack.

❹ Retrieve the i'th SV from the stack.

❺ Now that we have the SV* from the stack, retrieve the associated integer and add it into the sum.

Rather than use the `T_ARRAY` typemap entry to do this (which would require us to provide a memory allocator as well as losing information about the size of the array) we have written this using `Inline`'s stack macros. Since the set of macros is limited they are designed to be very simple and easy to understand rather than having to face the daunting contents of the perl internals documentation. Using them has the additional advantage that you are not tied to the perl XS macros themselves - if XS is changed or replaced then it is likely that your `Inline` module would continue working without any problems.

7.3.2.3. Multiple return arguments

Just as we can read multiple arguments off the stack we can also push return arguments onto the stack using `Inline` macros. Here is how we would implement the function to return the current time in a hash that was described earlier in Section 6.5.3.2:

```
void gmtime_as_list( time_t clock )                         ❶
{
  struct tm * tmbuf;
  Inline_Stack_Vars;                                        ❷
  Inline_Stack_Reset;                                       ❸

  tmbuf = gmtime( &clock );

  Inline_Stack_Push( sv_2mortal( newSVpv("sec", 3) ));      ❹
  Inline_Stack_Push( sv_2mortal( newSViv(tmbuf->tm_sec) ));
  Inline_Stack_Push( sv_2mortal( newSVpv("min", 3) ));
  Inline_Stack_Push( sv_2mortal( newSViv(tmbuf->tm_min) ));
  Inline_Stack_Push( sv_2mortal( newSVpv("hour", 4) ));
  Inline_Stack_Push( sv_2mortal( newSViv(tmbuf->tm_hour) ));
  Inline_Stack_Push( sv_2mortal( newSVpv("mday", 4) ));
```

```
  Inline_Stack_Push( sv_2mortal( newSViv(tmbuf->tm_mday) ));
  Inline_Stack_Push( sv_2mortal( newSVpv("mon", 3) ));
  Inline_Stack_Push( sv_2mortal( newSViv(tmbuf->tm_mon) ));
  Inline_Stack_Push( sv_2mortal( newSVpv("year", 4) ));
  Inline_Stack_Push( sv_2mortal( newSViv(tmbuf->tm_year) ));
  Inline_Stack_Push( sv_2mortal( newSVpv("wday", 4) ));
  Inline_Stack_Push( sv_2mortal( newSViv(tmbuf->tm_wday) ));
  Inline_Stack_Push( sv_2mortal( newSVpv("yday", 4) ));
  Inline_Stack_Push( sv_2mortal( newSViv(tmbuf->tm_yday) ));

  Inline_Stack_Done;                                          ❺
}
```

❶ Just as for XS we use a void return type when we are pushing arguments onto the stack ourselves.

❷ This initializes the stack variables used by the other macros.

❸ This must be used before any variables are pushed onto the stack. It resets the stack pointer to the beginning of the stack (rather than at the end of the input arguments).

❹ Push an SV onto the stack. This is completely equivalent to `XPUSHs`.

❺ Use this macro to indicate when all the necessary variables have been pushed onto the stack.

All we have done here is replace `PUSHs` with `Inline_Stack_Push`. It's as simple as that!

7.3.3. Summary

`Inline` is a very powerful addition to your armoury. With only a small loss in overall flexibility (much of which you won't miss) you can mix perl and C code without ever having to worry about makefiles and XS syntax. All you need is a knowledge of the variable manipulation API and possibly typemaps. The main things to be aware of are:

- You can not have input arguments that are also return arguments. If you need this simply use the SV* as an argument and modify it directly. For example,

  ```
  void modify_inplace( SV* sv) {
    sv_setiv(sv, 5);
  }
  ```

  If the above function is called via `Inline` then the scalar argument will be set to `5` on completion. Also, as mentioned in Section 2.6 in many cases the interface is better designed handling this in a different way.

- Be careful with memory allocation. You will not be able to explicitly free memory when you return from your `Inline` function (especially if you have allocated memory for a string buffer that is to be returned) so either just use an SV directly (either by pushing it on the stack or by returning it) or

allocate memory using mortal SV's (`sv_2mortal(NEWSV(..))` and return the buffer pointer with `SvPVX`) rather than with `New`.

We can not cover all of `Inline` here but hopefully this section has given a taste of what is possible.

* *If we want to flesh out this section we could include instructions on how to package these into modules (without installing Inline) and how to link against external libraries. In general I think that is probably overkill since this chapter exists to fill the reader with wonder at the possibilities - they can read up further themselves.*

7.4. PDL::PP

We saw in Section 6.6.1.3.2 that creating interfaces to from the Perl Data Language to external libraries using XS is quite complex. In addition to the native complexity of PDL there are four issues that further complicate PDL/XS interfaces:

Data typing

Many libraries have different routines for different data types. Writing XS interfaces that are identical except for the data type of the PDL is time consuming and error prone.

Slicing

When a subsection of a PDL is used PDL does not make a copy (data can be modified "in place"). If a slice is passed to XS the pointer will not be referring to the start of the slice but the start of the parent PDL!

PDL "threading"

One nice feature of PDL is its ability to automatically "thread" over additional dimensions. This has to be implemented in C for speed but is essentially impossible to get right if it is coded "by hand".

Changes in API

If the internal API for either PDL or Perl is modified it is highly likely that the XS code would have to be fixed. Writing XS code that works for multiple versions of internals API is difficuly and quickly leaves to an "ifdef forest" of C pre processor directives.

To solve the above issues a PDL pre-processor language was written to abstract out the numeric interface from the XS implementation. These interface definitions (using a file suffix of `.pd`) are automatically processed by the `PDL::PP` module to generate the required XS code. `PDL::PP` automatically generates code for multiple data types, keeps track of slicing and implement threading. If the API is changed all that needs to be modified is `PDL::PP`, PDL itself can be rebuilt with minimal changes.

In fact the primary goal of `PDL::PP` is to allow numerical code to be written in a C-like language (for speed) without having to worry about XS. Support for external libraries is a side effect of this goal. `PDL::PP` is an extremely complicated module and no attempt will be made to describe all it's features. What we will show is how to write a `PDL::PP` interface to the `sum` described in Chapter 6.

7.4.1. The .pd file

Instead of using a .xs file PDL::PP code is written to a .pd file. In this example we'll create a file called sum.pd to hold the definitions. This file is a perl program that is run as part of the make process (see the next section to see how). This program creates the normal XS infrastructure: the XS file and the associated perl module. For this example the first thing we need to do is to supply the code for the sum itself. We do this using the pp_addhdr . This function is used to place additional C code at the top of the output XS file (before the MODULE line). That C code is supplied as an argument:

```
pp_addhdr('

int sum ( int num, int * array ) {
  int thesum = 0;
  int count;
  for (count = 0; count < num; count++) {
    thesum += array[count];
  }
  return thesum;
}

');
```

Now that the C function is present we can supply the PP code for the PDL interface. We do this using the pp_def function:

```
pp_def( 'sum',                                          ❶
        Pars => 'a(n); [o]b()',                         ❷
        GenericTypes => [L],                            ❸
        Code => '$b() = sum( $SIZE(n), $P(a));',        ❹
);
```

❶ The first argument to pp_def is the name of the routine being generated.

❷ This is the calling signature of the PDL routine. Here we are saying that the first argument is a one dimensional vector of dimension n with an output argument (that can also be a second input argument treated as a buffer) that is simple scalar PDL (there are no dimensions specified).

❸ GenericTypes indicates to PP that only specific data types are supported by the routine. In this case only type long int is supported by our C function.

❹ This is the actual implementation of the PP routine. $SIZE(n) is used to retrieve the size of dimensions labelled n. $P(a) retrieves the pointer to the PDL named $a. These two arguments are passed to the C sum and the result is stored in the scalar variable named $b.

This all looks fairly strange at first but there is some logic to it all and it has successfully formed a layer between you and XS (the XS file generated from this example PP file is 30kB!). This definition on its own is useless so the next step is to convert the file to XS code[3].

7.4.2. The `Makefile.PL`

Since we now have a file called `sum.pd` rather than the file `Sum.xs` expected by `ExtUtils::MakeMaker` we have to add some PDL helper routines to the `Makefile.PL` to make sure that the PP definition file is processed in order to generate a normal XS file. We do this by adding a new makefile target and retrieving PDL-specific build options from the module `PDL::Core::Dev`:

```
# Use this as a template for the Makefile.PL for
# any external PDL module.

use ExtUtils::MakeMaker;
use PDL::Core::Dev qw/ pdlpp_stdargs pdlpp_postamble/;

@pack = ([qw/ sum.pd Sum PDL::Sum /]);
%hash = pdlpp_stdargs(@pack);

WriteMakefile(%hash);

sub MY::postamble {
        pdlpp_postamble(@::pack);
}  # Add genpp rule
```

Now if we attempt to build this module we get the following output (on linux):

```
% perl Makefile.PL
Writing Makefile for PDL::Sum
% make
/local/bin/perl -I/usr/local/perl-5.6/lib/site_perl/5.6.0/i686-linux/blib/lib
   -I/usr/local/perl-5.6/lib/site_perl/5.6.0/i686-linux/blib/arch
   "-MPDL::PP qw/PDL::Sum PDL::Sum Sum/" sum.pd
cp Sum.pm blib/lib/PDL/Sum.pm
/local/bin/perl -I/usr/local/perl-5.6/lib/5.6.0/i686-linux
  -I/usr/local/perl-5.6/lib/5.6.0
   /usr/local/perl-5.6/lib/5.6.0/ExtUtils/xsubpp
    -typemap /usr/local/perl-5.6/lib/5.6.0/ExtUtils/typemap
     -typemap/usr/local/perl-5.6/lib/site_perl/5.6.0/i686-linux/PDL/
Core/typemap.pdl
      Sum.xs > Sum.xsc && mv Sum.xsc Sum.c
gcc -c -I/usr/local/perl-5.6/lib/site_perl/5.6.0/i686-linux/PDL/Core
   -fno-strict-aliasing -D_LARGEFILE_SOURCE -D_FILE_OFFSET_BITS=64 -O2
   -DVERSION=\"0.10\" -DXS_VERSION=\"0.10\" -fpic
   -I/usr/local/perl-5.6/lib/5.6.0/i686-linux/CORE  Sum.c
Running Mkbootstrap for PDL::Sum ()
chmod 644 Sum.bs
LD_RUN_PATH="" gcc -o blib/arch/auto/PDL/Sum/Sum.so
   -shared -L/usr/local/lib Sum.o
chmod 755 blib/arch/auto/PDL/Sum/Sum.so
cp Sum.bs blib/arch/auto/PDL/Sum/Sum.bs
```

```
chmod 644 blib/arch/auto/PDL/Sum/Sum.bs
Manifying blib/man3/PDL::Sum.3
```

The first step in the build is now to run the pd file through perl before proceeding with a normal build (**xsubpp** followed by the C compiler).

7.4.3. Pure PDL

Now that we have shown how to call an external library us `PDL::PP` it may be instructive to show the equivalent routine written using the PP language without referring to any external C code:

```
pp_def(
       'sumover',
       Pars => 'a(n); [o]b();',
       Code => 'double tmp=0;                          ❶
               loop(n) %{                              ❷
                   tmp += $a();                        ❸
               %}                                      ❹
               $b() = tmp;                             ❺
               '
              );
```

❶ Initialise a new double precision variable.

❷ Loop over dimension `n` using the inbuilt `loop()` macro.

❸ Add the current value to the temporary placeholder. Note how no index is required. `PDL::PP` automatically works out what index you are referring to.

❹ This is the special syntax for ending a loop.

❺ Assign the total to the output variable.

This code is still translated to C and built as an XS module but now works on all data types natively. This is the recommended approach to writing fast loops in PDL and is one of the resons that PDL can approach the speed of pure C.

7.5. The Rest

The modules and packages described so far are by no means an exhaustive list but do cover the techniques you would most likely want to pursue. There are other, less portable, methods for calling functions in C shared libraries from perl but with the appearance of the `Inline` module most of these techniques can (and probably should) be safely ignored. For example, the `C::DynaLib` module can be

used to call C functions directly from shared libraries and did have a use for prototyping simple systems without requiring XS. The following example (from the `C::DynaLib` documentation) shows how to use it to call a function from the standard math library:

```
use C::DynaLib;
$libm = new C::DynaLib("-lm");
$sinh = $libm->DeclareSub("sinh", "d", "d");
print "The hyperbolic sine of 3 is ", &$sinh(3), "\n";
```

and this does print the expected answer of 10.018 (on supported architectues). With `Inline` this would be written as

```
use Inline "C";

print "The hyperbolic sine of 3 is ", mysinh(3), "\n";

__END__
__C__
double mysinh (double ang) {
    return sinh(ang);
}
```

which has the two key advantages of being simpler to write[4] and more portable. The PDL equivalent of `C::DynaLib` is `PDL::CallExt` and this has also been superceded, this time by `PDL::PP`.

7.6. Further reading

SWIG

http://www.swig.org

Inline

http://inline.perl.org

Notes

1. SWIG has undergone a major rewrite between 1999 and 2001 and version 1.3.6 was the first stable version released since February 1998 (when version 1.1p5 was released). The examples all used version 1.3.9.
2. At least, what is happening with version 0.43 of `Inline`. Development of this module is fairly rapid so things may have changed to some extent by the time you read this.
3. An `Inline::PDLPP` module is under development at this time. This will allow PP definitions to be placed in perl programs directly in the same way as described in Section 7.3 for the C language.
4. This can be written in 2 lines for those of you who prefer compactness:

```
use Inline C => "double mysinh (double ang) { return sinh(ang); }";
```

```
print "The hyperbolic sine of 3 is ", mysinh(3), "\n";
```

Chapter 8. Functions for Embedding and Internals

Chapter 9. Embedding Perl in C

9.1. What is Embedding?

In the first half of this book, we've looked at what it means to extend Perl with additional routines from C. Sometimes, however, there are occasions when you want to call a piece of Perl from inside a C program - we call this "embedding" Perl in C, because we link an entire Perl interpreter inside another C program.

9.1.1. When do I want to embed?

The best and most well-known example of embedding Perl in C is Apache's `mod_perl` module. This allows the user to interact with Perl at every level of the Apache web server - one can write configuration files in Perl, write Perl programs to handle HTTP requests with Perl objects, and so on. In short, it allows you to use Perl to script and control the rest of the program.

More specific examples include the embedding of Perl into the Xchat IRC client to enable the user to script complex actions in Perl; the GIMP graphics editor, which allows graphical manipulations to be encoded in Perl; vim, a text editor, which can be both configured and manipulated using Perl; and gnumeric, the GNOME spreadsheet, which exposes the data in the spreadsheet cells to Perl for additional manipulation.

All of these examples have some common objectives: to make the application extensible through user-provided plugin scripts, to make configuration more flexible by involving a high-level language and the control structures that it provides, and to help the user script common or complex sequences of actions. If an application that you are working with could benefit from these features, you should contemplate the possibility of embedding a Perl interpreter.

9.1.2. When do I *not* want to embed?

Embedding Perl into a program is not a panacea, and embedding Perl into an existing program is not a step to be taken lightly. We don't recommended embedded Perl as a cheap way of avoiding writing a configuration parser or extensible scripting system.

Another thing to be conscious of is that embedding Perl into an application will increase its size and memory usage, possibly introduce memory leaks or instabilities, and occasionally slow the application down. Nevertheless, the examples we have given above do show that there is sometimes a really big gain to be won by including Perl in your program.

9.1.3. Things to think about

In the next chapter, we'll look in more detail at the decisions that need to be made when embedding Perl into an application. Fundamentally, however, you need to consider to what degree Perl should have access to the guts of your program.

This in turn influences details such as which data structures you are going to expose to Perl and how they will appear to the Perl programmer; what C functions in your API will be available, and, again, how they would be used from Perl; where your Perl programs will come from and at what point in the program they will be used, and so on.

Again, we'll see practical answers to these questions in the next chapter; let's now see an example of calling a Perl program from inside a C program.

9.2. "Hello C" from Perl

The fundamentals of embedding are simple: we perform almost exactly the same function as the main body of the `perl` binary. That's to say, we construct and initialize an interpreter, use it to parse a string of code, and then execute that code. Here's a simple program which does that.

```
#include <EXTERN.h>                                          ❶
#include <perl.h>                                            ❷

static PerlInterpreter *my_perl;                             ❸

int main(int argc, char **argv)
{
    char* command_line[] = {"", "-e", "print \"Hello from C!\\n\";"};❹

    my_perl = perl_alloc();                                  ❺
    perl_construct(my_perl);                                 ❻
    perl_parse(my_perl, NULL, 3, command_line, (char **)NULL);❼
    perl_run(my_perl);                                       ❽
    perl_destruct(my_perl);                                  ❾
    perl_free(my_perl);                                      (10)
}
```

❶ This first header file sets up some macros to tell the main Perl header file that we're not really Perl itself, but an external application using the Perl API.

❷ Now we load up the main Perl header, which provides the macros and prototypes for all the functions in the Perl API.

❸ Since one can actually have multiple Perl interpreters in a program, all the interpreter-global data gets stored in a structure called, naturally, a PerlInterpreter.

❹ Even when we're embedding Perl, we're dealing with the honest-to-goodness Perl interpreter, which expects to get arguments from the command line. Hence, we have to provide a set of command line arguments, just as we'd expect to find in `argv`. (And just like in `argv`, the first element of the array is the name of the command, rather than the first command line argument; since we're not bothered about that, we leave it blank.)

❺ Just like any other structure pointer, we need to allocate memory for it; `perl_malloc` returns us some memory for a Perl interpreter.

❻ Next, we set up the interpreter and all its associated data structures with `perl_construct`.

❼ Now we're in a position where we can parse the incoming Perl "script", which is specified in the `-e` argument to the faked command line.

❽ As we know, Perl is bytecode-compiled - first, the code is parsed into an internal representation, and then it's run; `perl_run` starts the main loop running the code.

❾ Finally, we cleanly shut down the interpreter, and release the memory that had been allocated for it.

It's interesting to compare this with the source to the Perl interpreter itself; if you don't believe we're performing the same functions as the Perl interpreter, take a look at the guts of `miniperlmain.c`:

```
if (!PL_do_undump) {
    my_perl = perl_alloc();
    if (!my_perl)
        exit(1);
    perl_construct(my_perl);
    PL_perl_destruct_level = 0;
}

exitstatus = perl_parse(my_perl, xs_init, argc, argv, (char **)NULL);
if (!exitstatus) {
    exitstatus = perl_run(my_perl);
   perl_destruct(my_perl);
   exitstatus = STATUS_NATIVE_EXPORT;
} else {
  perl_destruct(my_perl);
}

perl_free(my_perl);

PERL_SYS_TERM();

exit(exitstatus);
return exitstatus;
```

As you can see, this is more or less the same as the above with a little more error checking.

9.2.1. Compiling Embedded Programs

Compiling programs which embed Perl is a little bit special; you need to ensure that you're compiling the code with exactly the same options that were used to compile the Perl interpreter. While we could get these out of `Config.pm`, there's a module which makes it very easy for us: `ExtUtils::Embed`.

As we know from the first chapter, a program's compilation takes place in two stages: compilation proper, and linking. `ExtUtils::Embed` provides two functions, `ccopts` and `ldopts` to tell us the options for each stage. In fact, if we run these functions from the Perl command line, they'll handily spit the options to standard output, making it ideal to use as part of our build process.

So, let's take the example code above, `simple.c`, and compile it and then link it:

```
 % cc -o simple.o -c simple.c `perl -MExtUtils::Embed -e ccopts`
 % cc -o simple simple.o `perl -MExtUtils::Embed -e ldopts`
 % ./simple
Hello from C!
```

Now we have a way to execute simple Perl programs from C, albeit if we specify them on the Perl command line; let's make this a bit more powerful.

9.3. Passing Data

In a real embedded application, we need a way to share data between Perl and C. For instance, the Apache embedded Perl module, `mod_perl`, allows you to store configuration data in Perl variables.

As we saw in Section 5.2.2.1, Perl provides a function called `get_sv` which allows to grab an SV from the Perl symbol table. Let's suppose we're writing a mail client, (we'll call it "Hermes") and we want our users to be able to set some of the configuration in Perl. First, we'll look at general settings which apply to the whole application; in the next section, we'll write some logic for settings which apply on a per-message basis.

Our sample configuration file would look something like this:

```
package Hermes;

$save_outgoing = 1;

# Prefer vim, but use emacs if vim not available.
$editor = `which vim` || `which emacs`;
```

```
$quote_char = "> ";

$check_every = 10; # seconds
```

9.4. Calling Perl Routines

9.5. Using Perl Regular Expressions

9.6. Using C in Perl in C

Chapter 10. Embedding Case Study

Chapter 11. Introduction to Perl Internals

11.1. The Source Tree

This section introduces you to the major parts of the Perl source tree that you should familiarise yourself with.

11.1.1. The Perl Library

The most approachable part of the source code, for Perl programmers, is the Perl library. This lives in `lib/`, and comprises all the standard, pure Perl modules and pragmata that ship with `perl`.

There are both Perl 5 modules and unmaintained Perl 4 libraries, shipped for backwards compatibility. In Perl 5.6.0 and above, the Unicode tables are placed in `lib/unicode`.

11.1.2. The XS Library

In `ext/`, we find the XS modules which ship with Perl. For instance, the Perl compiler `B` can be found here, as can the DBM interfaces. The most important XS module here is `DynaLoader`, the dynamic loading interface which allows the runtime loading of every other XS module.

As a special exception, the XS code to the methods in the `UNIVERSAL` class can be found in `universal.c`.

11.1.3. The IO Subsystem

Recent versions of Perl come with a completely new standard IO implementation, `PerlIO`. This allows for several "layers" to be defined through which all IO is filtered, similar to the line disciplines mechanism in `sfio`. These layers interact with modules such as `PerlIO::Scalar`, also in the `ext/` directory.

The IO subsystem is implemented in `perlio.c` and `perlio.h`. Declarations for defining the layers are in `perliol.h`, and documentation on how to create layers is in `pod/perliol.pod`.

Perl may be compiled without `PerlIO` support, in which case there are a number of abstraction layers to present a unified IO interface to the Perl core. `perlsdio.h` aliases ordinary standard IO functions to their `PerlIO` names, and `perlsfio.h` does the same thing for the alternate IO library `sfio`.

The other abstraction layer is the "Perl host" scheme in `iperlsys.h`. This is confusing. The idea is to reduce process overhead on Win32 systems by having multiple Perl interpreters access all system calls through a shared "Perl host" abstraction object. There is an explanation of it in `perl.h`, but it is best avoided.

11.1.4. The Regexp Engine

Another area of the Perl source best avoided is the regular expression engine. This lives in `re*.*`. The regular expression matching engine is, roughly speaking, a state machine generator. Your match pattern is turned into a state machine made up of various match nodes - you can see these nodes in `regcomp.sym`. The compilation phase is handled by `regcomp.c`, and the state machine's execution is performed in `regexec.c`.

11.1.5. The Parser and Tokeniser

The first stage in Perl's operation is to "understand" your program. This is done by a joint effort of the tokeniser and the parser. The tokeniser is found in `toke.c`, and the parser in `perly.c`. (although you're far, far better off looking at the YACC source in `perly.y`)

The job of the tokeniser is to split up the input into meaningful chunks, or *tokens*, and also to determine what type of thing they represent - a Perl keyword, a variable, a subroutine name, and so on. The job of the parser is to take these tokens and turn them into "sentences", understanding their relative meaning in context.

11.1.6. Variable Handling

As we already know, Perl provides C-level implementations of scalars, arrays and hashes. The code for handling arrays is in `av.*`, hashes are in `hv.*` and scalars are in `sv.*`.

11.1.7. Run-time Execution

What about the code to Perl's built-ins - `print`, `foreach` and the like? These live in `pp.*`, and some of the functionality is shelled out to `doio.c`.

The actual main loop of the interpreter is in `run.c`.

* *Following inserted from netthink documentation*

11.2. The Parser

Perl is a bytecode-compiled language, which means that execution of a Perl program happens in two stages. First of all, the program is read, parsed and compiled into an internal representation of the operations to be performed; after that, the interpreter takes over and traverses this internal representation, executing the operations in order. We'll first look at how Perl is parsed, before moving on to looking at the internal representation of a program.

As discussed above the parser lives in `perly.y`. This is code in a language called Yacc, which is converted to C using the **byacc** command. In order to understand this language, we need to understand how grammars work and how parsing works.

> In fact, Perl needs to do some fixing up on the **byacc** output to have it deal with dynamic rather than static memory allocation. Hence, if you make any changes to `perly.y`, just running **byacc** isn't enough - you need to run the Make target `run_byacc`, which will do the fixups that Perl requires.

11.2.1. BNF and Parsing

Computer programmers define a language by its grammar, which is a set of rules. They usually describe this grammar in a form called "Backhaus-Naur Form" [1] or *BNF*. BNF tells us how phrases fit together to make sentences. For instance, here's a simple BNF for English - obviously, this isn't going to describe the whole of the English grammar, but it's a start:

```
sentence   : nounphrase verbphrase nounphrase;

verbphrase : VERB;

nounphrase : NOUN
| ADJECTIVE  nounphrase
| PRONOMINAL nounphrase
| ARTICLE    nounphrase;
```

Here is the prime rule of BNF: you can make the thing on the left of the colon if you see all the things on the right in sequence. So, this grammar tells us that a sentence is made up of a noun phrase, a verb phrase and then a noun phrase. The vertical bar does exactly what it does in regular expressions: you can make a noun phrase if you have a noun, or an adjective plus another noun phrase, or an article plus a noun phrase. Turning the things on the right into the thing on the left is called a *reduction*. The idea of parsing is to reduce all of the input down to the first thing in the grammar - a sentence.

You'll notice that things which can't be broken down any further are in capitals - there's no rule which tells us how to make a noun, for instance. This is because these are fed to us by the lexer; these are called *terminal symbols*, and the things which aren't in capitals are called *non-terminal symbols*. Why? Well, let's see what happens if we try and parse a sentence in this grammar.

The text right at the bottom - "my cat eats fish" - is what we get in from the user. The tokeniser then turns that into a series of tokens - "PRONOMINAL NOUN VERB NOUN". From that, we can start performing some reductions: we have a pronominal, so we're looking for a noun phrase to satisfy the `nounphrase : PRONOMINAL nounphrase` rule. Can we make a noun phrase? Yes, we can, by reducing the `NOUN` ("cat") into a `nounphrase`. Then we can use `PRONOMINAL nounphrase` to make another `nounphrase`.

Now we've got a `nounphrase` and a `VERB`. We can't do anything further with the `nounphrase`, so we'll switch to the `VERB`, and the only thing we can do with that is turn it into a `verbphrase`. Finally, we can reduce the noun to a `nounphrase`, leaving us with `nounphrase verbphrase nounphrase`. Since we can turn this into a `sentence`, we've parsed the text.

11.2.2. Parse actions and token values

It's important to note that the tree we've constructed above - the "parse tree" - is only a device to help us understand the parsing process. It doesn't actually exist as a data structure anywhere in the parser. This is actually a little inconvenient, because the whole point of parsing a piece of Perl text is to come up with a data structure pretty much like that.

Not a problem. Yacc allows us to extend BNF by adding actions to rules - every time the parser performs a reduction using a rule, it can trigger a piece of C code to be executed. Here's an extract from Perl's grammar in `perly.y`:

```
term    :   term ASSIGNOP term
  { $$ = newASSIGNOP(OPf_STACKED, $1, $2, $3); }
        |   term ADDOP term
  { $$ = newBINOP($2, 0, scalar($1), scalar($3)); }
```

The pieces of code in the curlies are actions to be performed. Here's the final piece of the puzzle: each symbol carries some additional information around. For instance, in our "cat" example, the first `NOUN` had the value "cat". You can get at the value of a symbol by a Yacc variable starting with a dollar sign: in the example above, `$1` is the value of the first symbol on the right of the colon (`term`), `$2` is the value of the second symbol (either `ASSIGNOP` or `ADDOP` depending on which line you're reading) and so on. `$$` is the value of the symbol on the left. Hence information is propagated "up" the parse tree by manipulating the information on the right and assigning it to the symbol on the left.

11.2.3. Parsing some Perl

So, let's see what happens if we parse the Perl code **`$a = $b + $c`**. We have to assume that **`$a`**, **`$b`** and **`$c`** have already been parsed a little; they'll turn into `term` symbols. Each of these symbols will have a value, and that will be an "op". An "op" is a data structure representing an operation, and the operation to be represented will be that of retrieving the storage pointed to by the appropriate variable.

Let's start from the right[2] , and deal with **`$b + $c`**. The **`+`** is turned by the lexer into the terminal symbol `ADDOP`. Now, just like there can be lots of different nouns that all get tokenised to `NOUN`, there can be several different `ADDOP`s - concatenation is classified as an `ADDOP`, so **`$b . $c`** would look just the same to the parser. The difference, of course, is the value of the symbol - this `ADDOP` will have the value `'+'`.

Hence, we have `term ADDOP term`. This means we can perform a reduction, using the second rule in our snippet. When we do that, we have to perform the code in curlies underneath the rule - `{ $$ = newBINOP($2, 0, scalar($1), scalar($3)); }`. `newBINOP` is a function which creates a new binary "op". The first argument is the type of binary operator, and we feed it the value of the second symbol. This is `ADDOP`, and as we have just noted, this symbol will have the value `'+'`. So although `'.'` and `'+'` look the same to the parser, they'll eventually be distinguished by the value of their symbol. Back to `newBINOP`. The next argument is the flags we wish to pass to the op. We don't want anything special, so we pass zero.

Then we have our arguments to the binary operator - obviously, these are the value of the symbol on the left and the value of the symbol on the right of the operator. As we mentioned above, these are both "op"s, to retrieve the values of `$b` and `$c` respectively. We assign the new "op" created by `newBINOP` to be the value of the symbol we're propagating upwards. Hence, we've taken two ops - the ones for `$b` and `$c` - plus an addition symbol, and turned them into a new op representing the combined action of fetching the values of `$b` and `$c` and then adding them together.

Now we do the same thing with `$a = ($b+$c)`. I've put the right hand side in brackets to show that we've already got something which represents fetching `$b` and `$c` and adding them. `=` is turned into an `ASSIGNOP` by the tokeniser in the same way as we turned `+` into an `ADDOP`. And, in just the same way, there are various different types of assignment operator - `||=` and `&&=` are also passed as `ASSIGNOP`s. From here, it's easy: we take the `term` representing `$a`, plus the `ASSIGNOP`, plus the `term` we've just constructed, reduce them all to another `term`, and perform the action underneath the rule. In the end, we end up with a data structure a little like this:

You can find a hypertext version of the Perl grammar at http://simon-cozens.org/hacks/grammar.pdf

11.3. The Tokeniser

The tokeniser, in `toke.c` is one of the most difficult parts of the Perl core to understand; this is primarily because there is no real "roadmap" to explain its operation. In this section, we'll try to show how the tokeniser is put together.

11.3.1. Basic tokenising

The core of the tokeniser is the intimidatingly long `yylex` function. This is the function called by the

parser, `yyparse`, when it requests a new token of input.

First, some basics. When a token has been identified, it is placed in `PL_tokenbuf`. The file handle from which input is being read is `PL_rsfp`. The current position in the input is stored in the variable `PL_bufptr`, which is a pointer into the PV of the SV `PL_linestr`. When scanning for a token, the variable `s` advances from the start of `PL_bufptr` towards the end of the buffer (`PL_bufend`) until it finds a token.

The first thing the parser does is test whether the next thing in the input stream has already been identified as an identifier; when the tokeniser sees `'%'`, `'$'` and the like as part of the input, it tests to see whether it introduces a variable. If so, it puts the variable name into the token buffer. It then returns the type sigil (`%`, `$`, etc.) as a token, and sets a flag (`PL_pending_ident`) so that the next time `yylex` is called, it can pull the variable name straight out of the token buffer. Hence, right at the top of `yylex`, you'll see code which tests `PL_pending_ident` and deals with the variable name.

11.3.1.1. Tokeniser State

Next, if there's no identifier in the token buffer, it checks its tokeniser state. The tokeniser uses the variable `PL_lex_state` to store state information.

One important state is `LEX_KNOWNEXT`, which occurs when Perl has had to look ahead one token to identify something. If this happens, it has tokenised not just the next token, but the one after as well. Hence, it sets `LEX_KNOWNEXT` to say "we've already tokenised this token, simply return it."

The functions which set `LEX_KNOWNEXT` are `force_word`, which declares that the next token has to be a word, (for instance, after having seen an arrow in `$foo->bar`) `force_ident`, which makes the next token an identifier, (for instance, if it sees a `*` when not expecting an operator, this must be a glob) `force_version`, (on seeing a number after `use`) and the general `force_next`.

Many of the other states are to do with interpolation of double-quoted strings; we'll look at those in more detail in the next section.

11.3.1.2. Looking ahead

After checking the lexer state, it's time to actually peek at the buffer and see what's waiting; this is the start of the giant `switch` statement in the middle of `yylex`, just following the label `retry`.

One of the first things we check for is character zero - this signifies either the start or the end of the file or the end of the line. If it's the end of the file, the tokeniser returns zero and the game is one; at the beginning of the file, Perl has to process the code for command line switches such as `-n` and `-p`. Otherwise, Perl calls `filter_gets` to get a new line from the file through the source filter system, and calls `incline` to increase the line number.

The next test is for comments and new lines, which Perl skips over. After that come the tests for individual special characters. For instance, the first test is for minus, which could be unary minus if followed by a number or identifier, or the binary minus operator if Perl is expecting an operator, or the arrow operator if followed by a `>`, or the start of a filetest operator if followed by an appropriate letter, or a quoting option such as `(-foo => "bar" )`. Perl tests for each case, and returns the token type using one of the upper-case token macros defined at the beginning of `toke.c`: `OPERATOR`, `TERM`, and so on.

If the next character isn't a symbol that Perl knows about, it's an alphabetic character which might start a keyword: the tokeniser jumps to the label `keylookup` where it checks for labels and things like `CORE::function`. It then calls `keyword` to test whether it is a valid built-in or not - if so, `keyword` turns it into a special constant (such as `KEY_open`) which can be fed into the `switch` statement. If it's not a keyword, Perl has to determine whether it's a bareword, a function call or an indirect object or method call.

11.3.1.3. Keywords

The final section of the `switch` statement deals with the `KEY_` constants handed back from `keyword`, performing any actions necessary for using the builtins. (For instance, given `__DATA__`, the tokeniser sets up the `DATA` filehandle.)

11.3.2. Sublexing

"Sublexing" refers to the the fact that inside double-quoted strings and other interpolation contexts (regular expressions, for instance) a different type of tokenisation is needed.

This is typically started after a call to `scan_str`, which is an exceptionally clever piece of code which extracts a string with balanced delimiters, placing it into the SV `PL_lex_stuff`. Then `sublex_start` is called which sets up the data structures used for sublexing and changes the lexer's state to `LEX_INTERPPUSH`, which is essentially a scoping operator for sublexing.

Why does sublexing need scoping? Well, consider something like `"Foo\u\LB\uarBaz"`. This actually gets tokenized as the moral equivalent of `"Foo" . ucfirst(lc("B" . ucfirst("arBaz")))`. The push state (which makes a call to `sublex_push`) quite literally pushes an opening bracket onto the input stream.

This in turn changes the state to `LEX_INTERPCONCAT`; the concatentation state uses `scan_const` to pull out constant strings and supplies the concatenation operator between them. If a variable to be interpolated is found, the state is changed to `LEX_INTERPSTART`: this means that `"foo$bar"` is changed into `"foo".$bar` and `"foo@bar"` is turned into `"foo".join($",@bar)`.

There are times when it is not sure when sublexing of an interpolated variable should end - in these

cases, the horrifyingly scary function `intuit_more` is called to make an educated guess on the likelihood of more interpolation.

Finally, once sublexing is done, the state is set to `LEX_INTERPEND` which fixes up the closing brackets.

11.4. Summary

So far, we've briefly examined how Perl turns Perl source input into a tree data structure suitable for executing; next, we'll look more specifically at the nature of the nodes in that tree.

There are two stages to this operation: the tokeniser, `toke.c`, chops up the incoming program and recognises different token types; the parser `perly.y` then assembles these tokens into phrases and sentences. In reality, the whole task is driver by the parser - Perl calls `yyparse` to parse a program, and when the parser needs to know about the next token, it calls `yylex`.

While the parser is relatively straightforward, the tokeniser is somewhat more tricky. The key to understanding it is to divide its operation into checking tokeniser state, dealing with non-alphanumeric symbols in ordinary program code, dealing with alphanumerics, and dealing with double-quoted strings and other interpolation contexts.

Very few people actually understand the whole of how the tokeniser and parser work, but this chapter should have given you a useful insight into how Perl understands program code, and how to locate the source of particular behaviour inside the parsing system.

* *Ops is from netthink*

11.5. Op Code Trees

So we've seen that the job of the parsing stage is to reduce a program to a tree structure, and each node of the tree represents an operation. In this chapter, we'll look more closely at those operations: what they are, how they're coded, and how they fit together.

11.5.1. The basic op

Just AVs and HVs are "extensions" of the basic SV structure, there are a number of different "flavours" of ops, built on a basic OP structure; you can find this structure defined as `BASEOP` in `op.h`:

```
OP*         op_next;
OP*         op_sibling;
OP*         (CPERLscope(*op_ppaddr))(pTHX);
```

```
    PADOFFSET   op_targ;
    OPCODE      op_type;
    U16         op_seq;
    U8          op_flags;
   U8          op_private;
```

Some of these fields are easy to explain, so we'll deal with them now.

The `op_next` field is a pointer to the next op which needs to be executed. We'll see later, in , how the "thread of execution" is derived from the tree.

`op_ppaddr` is the address of the C function which carries out this particular operation. It's stored here so that our main execution code can simply dereference the function pointer and jump to it, instead of having to perform a lookup.

Each unique operation has a different number; this can be found in the `enum` in `opnames.h`:

```
typedef enum opcode {
    OP_NULL,        /* 0 */
    OP_STUB,        /* 1 */
    OP_SCALAR,      /* 2 */
    OP_PUSHMARK,    /* 3 */
    OP_WANTARRAY,   /* 4 */
    OP_CONST,       /* 5 */
    OP_GVSV,        /* 6 */
    OP_GV,          /* 7 */
    ...
};
```

The number of the operation to perform is stored in the `op_type` field. We'll examine some of the more interesting operations in .

`op_flags` is a set of flags generic to all ops; `op_private` stores flags which are specific to the type of op. For instance, the `repeat` op which implements the `x` operator has the flag `OPpREPEAT_DOLIST` set when it's repeating a list rather than a string. This flag only makes sense for that particular operation, so is stored in `op_private`. Private flags have the `OPp` prefix, and public flags begin with `OPf`.

`op_seq` is a sequence number allocated by the optimizer. It allows for, for instance, correct scoping of lexical variables by storing the sequence numbers of the beginning and end of scope operations inside the pad.

As for the remaining fields, we'll examine `op_sibling` in and `op_targ` in

11.5.1.1. The different operations

Perl has currently 351 different operations, implementing all the built-in functions and operators, as well as the more structural operations required internally - entering and leaving a scope, compiling regular expressions and so on.

The array `PL_op_desc` in `opcode.h` describes each operation: it may be easier to follow the data from which this table is generated, at the end of `opcode.pl`. We'll take a longer look at that file later on in this chapter.

Many of the operators are familiar from Perl-space, such as `concat` and `splice`, but some are used purely internally: for instance, one of the most common, `gvsv` fetches a scalar variable; `enter` and `leave` are block control operators, and so on.

11.5.1.2. Different "flavours" of op

There are a number of different "flavours" of op structure, related to the arguments of an operator and how it fits together with other ops in the op tree. For instance, `scalar` is a unary operator, a UNOP. This extends the basic op structure above with a link to the argument:

```
struct unop {
    BASEOP
    OP *    op_first;
};
```

Binary operators, such as `i_add`, (integer addition) have both a `first` and a `last`:

```
struct binop {
    BASEOP
    OP *    op_first;
    OP *    op_last;
};
```

List operators are more interesting; they too have a `first` and a `last`, but they also have some ops in the middle, too. This is where `op_sibling` above comes in; it connects ops "sibling" ops on the same level in a list. For instance, look at the following code and the graph of its op tree:

```
open FILE, "foo";
print FILE "hi\n";
close FILE;
```

The dashed lines represent `op_sibling` connections. The root operator of every program is the list operator `leave`, and its children are the statements in the program, separated by `nextstate` (next statement) operators. `open` is also a list operator, as is `print`. The first child of `print` is `pushmark`, which puts a mark on the stack (see) so that Perl knows how many arguments on the stack belong to `print`. The `rv2gv` turns a reference to the filehandle `FILE` into a GV, so that `print` can print to it, and the final child is the constant `"hi\n"`.

Some operators hold information about the program; these are COPs, or "code operators". Their definition is in `cop.h`:

```
struct cop {
    BASEOP
    char *  cop_label;  /* label for this construct */
#ifdef USE_ITHREADS
    char *  cop_stashpv;    /* package line was compiled in */
    char *  cop_file;   /* file name the following line # is from */
#else
    HV *    cop_stash;  /* package line was compiled in */
    GV *    cop_filegv; /* file the following line # is from */
#endif
    U32     cop_seq;    /* parse sequence number */
    I32     cop_arybase;    /* array base this line was compiled with */
    line_t      cop_line;       /* line # of this command */
    SV *    cop_warnings;   /* lexical warnings bitmask */
    SV *    cop_io;     /* lexical IO defaults */
};
```

COPs are inserted between every statement; they contain the label (for `goto`, `next` and so on) of the statement, the file name, package and line number of the statement and lexical hints such as the current value of `$[`, warnings and IO settings. Note that this doesn’t contain the current CV or the padlist - these are kept on a special stack called the "context stack".

The final type of op is the null op: any op with type zero means that a previous op has been optimized away; we’ll look at how this is done later in this chapter, but for now, you should skip over the null op when you see it in op trees.

11.5.1.3. Tying it all together

We’ve so far seen a little of how the op tree is connected together with `op_first`, `op_last`, `op_sibling`, and so on. Now we’ll look at how the tree gets manufactured, as how it gets executed.

11.5.1.3.1. "Tree" order

After our investigation of the parser in the previous chapter, it should now be straightforward to see how the op tree is created. The parser calls routines in `op.c` which create the op structures, passing ops further

"down" the parse tree as arguments. This threads together a tree as shown in the diagram above. For comparison, here is the what the example in that chapter (`$a = $b + $c`) really looks like as an op tree:

Again, you can see the places where an op was optimized away and became a null op. This is not so different from the simplified version we gave earlier.

11.5.1.3.2. Execution Order

The second thread through the op tree, indicated by the dotted line in our diagrams, is the execution order. This is the order in which Perl must actually perform the operations in order to run the program. The main loop of Perl is very, very simple, and you can see it in `run.c`:

```
while ((PL_op = CALL_FPTR(PL_op->op_ppaddr)(aTHX))) {
    PERL_ASYNC_CHECK();
}
```

That's it. That's all the Perl interpreter is. `PL_op` represents the op that's currently being executed. Perl calls the function pointer for that op and expects another op to be returned; this return value is then set to `PL_op`, which is executed in turn. Since everything apart from conditional operators (for obvious reasons) just return `PL_op->op_next`, the execution order through a program can be found by chasing the trail of `op_next` pointers from the start node to the root.

We can trace the execution order in several ways: if Perl is built with debugging, then we can say

```
perl -Dt -e 'open ...'
```

Alternatively, and perhaps more simply, the compiler module `B::Terse` (see) has an option to print the execution order, `-exec`. For instance, in our "open-print-close" example above, the execution order is:

```
% perl -MO=Terse,-exec -e 'open FILE, "foo"; ...'
OP (0x8111510) enter
COP (0x81121c8) nextstate
OP (0x8186f30) pushmark
SVOP (0x8186fe0) gv  GV (0x8111bd8) *FILE
SVOP (0x8186f10) const  PV (0x810dd98) "foo"
LISTOP (0x810a170) open [1]
COP (0x81114d0) nextstate
OP (0x81114b0) pushmark
SVOP (0x8118318) gv  GV (0x8111bd8) *FILE
UNOP (0x8111468) rv2gv
SVOP (0x8111448) const  PV (0x8111bfc) "hi\n"
LISTOP (0x8111488) print
COP (0x8111fe0) nextstate
SVOP (0x8111fc0) gv  GV (0x8111bd8) *FILE
UNOP (0x8111fa0) close
LISTOP (0x8111420) leave [1]
```

This program, just like every other program, starts with the `enter` and `nextstate` ops to enter a scope and begin a new statement respectively. Then a mark is placed on the argument stack: marks represent the start of a set of arguments, and a list operator can retrieve all the arguments by pushing values off the stack until it finds a mark. Hence, we're notifying Perl of the beginning of the arguments to the `open` operator.

The arguments in this case are merely the file handle to be opened and the file name; after operators put these two arguments on the stack, `open` can be called. This is the end of the first statement.

Next, the arguments to `print` begin. This is slightly more tricky, because while `open` can only take a true filehandle, `print` may take any sort of reference. Hence, `gv` returns the GV and then this is turned into the appropriate filehandle type by the `rv2gv` operator. After the filehandle come the arguments to be printed; in this case, a constant (`"hi\n"`). Now all the arguments have been placed on the stack, `print` can be called. This is the end of the second statement.

Finally, a filehandle is put on the stack and closed. Note that at this point, the connections between the operators - unary, binary, etc. - are not important; all manipulation of values comes not by looking at the children of the operators but by looking at the stack. The types of op are important for the construction of the tree in "tree order", but the stack and the `op_next` pointers are the only important things for the execution of the tree in execution order.

How is the execution order determined? The function `linklist` in `op.c` takes care of threading the `op_next` pointers in prefix order. It does so by recursively applying the following rule:

- If there is a child for the current operator, visit the child first, then its siblings, then the current op.

Hence, the starting operator is always the first child of the root operator, (always `enter`) the second op to be executed is its sibling, `nextstate`, and then the children of the next op are visited. Similarly, the root itself (`leave`) is always the last operator to be executed. Null operators are skipped over during optimization.

11.5.2. PP Code

We know the order of execution of the operations, and what some of them do. Now it's time to look at how they're actually implemented - the source code inside the interpreter that actually carries out `print`, `+`, and other operations.

The functions which implement operations are known as "PP Code" - "Push / Pop Code" - because most of their work involves popping off elements from a stack, performing some operation on it, and then

pushing the result back. PP code can be found in several files: `pp_hot.c` contains frequently used code, put into a single object to encourage CPU caching; `pp_ctl.c` contains operations related to flow control; `pp_sys.c` contains the system-specific operations such as file and network handling; `pack` and `unpack` recently moved to `pp_pack.c`, and `pp.c` contains everything else.

11.5.2.1. The argument stack

We've already talked a little about the argument stack. The Perl interpreter makes use of several stacks, but the argument stack is the main one.

The best way to see how the argument stack is used is to watch it in operation. With a debugging build of Perl, the `-Ds` command line switch prints out the contents of the stack in symbolic format between operations. Here is a portion of the output of running **`$a=5; $b=10; print $a+$b;`**:

```
(-e:1)  nextstate
    =>
(-e:1)  pushmark
    =>  *
(-e:1)  gvsv(main::a)
    =>  *  IV(5)
(-e:1)  gvsv(main::b)
    =>  *  IV(5)  IV(10)
(-e:1)  add
    =>  *  IV(15)
(-e:1)  print
    =>  SV_YES
```

At the beginning of a statement, the stack is typically empty. First, Perl pushes a mark onto the stack to know when to stop pushing off arguments for `print`. Next, the values of `$a` and `$b` are retrieved and pushed onto the stack.

The addition operator is a binary operator, and hence, logically, it takes two values off the stack, adds them together and puts the result back onto the stack. Finally, `print` takes all of the values off the stack up to the previous bookmark and prints them out. Let's not forget that `print` itself has a return value, the true value `SV_YES` which it pushes back onto the stack.

11.5.2.2. Stack manipulation

Let's now take a look at one of the PP functions, the integer addition function `pp_i_add`. The code may look formidable, but it's a good example of how the PP functions manipulate values on the stack.

```
PP(pp_i_add)                                                    ❶
{
    dSP; dATARGET; tryAMAGICbin(add,opASSIGN);                  ❷
    {
      dPOPTOPiirl_ul;                                           ❸
```

```
      SETi( left + right );                                    ❹
      RETURN;                                                  ❺
    }
}
```

❶ In case you haven't guessed, *everything* in this function is a macro. This first line declares the function `pp_i_add` to be the appropriate type for a PP function.

❷ Since following macros will need to manipulate the stack, the first thing we need is a local copy of the stack pointer, `SP`. And since this is C, we need to declare this in advance: `dSP` declares a stack pointer. Then we need an SV to hold the return value, a "target". This is declared with `dATARGET`; see for more on how targets work. Finally, there is a chance that the addition operator has been overloaded using the `overload` pragma. The `tryAMAGICbin` macro tests to see if it is appropriate to perform "A" (overload) magic on either of the scalars in a binary operation, and if so, does the addition using a magic method call.

❸ We will deal with two values, `left` and `right`. The `dPOPTOPiirl_ul` macro pops two SVs off the top of the stack, converts them to two integers (hence `ii`) and stores them into automatic variables `right` and `left`. (hence `rl`)

> The `_ul`? Look up the definition in `pp.h` and work it out...

❹ We add the two values together, and set the integer value of the target to the result, pushing the target to the top of the stack.

❺ As mentioned above, operators are expected to return the next op to be executed, and in most cases this is simply the value of `op_next`. Hence `RETURN` performs a normal return, copying our local stack pointer `SP` which we obtained above back into the global stack pointer variable, and then returning the `op_next`.

As you might have guessed, there are a number of macros for controlling what happens to the stack; these can be found in `pp.h`. The more common of these are:

`POPs`

Pop an SV off the stack and return it.

`POPpx`

Pop a string off the stack and return it. (Note: requires a variable "STRLEN `n_a`" to be in scope.)

`POPn`

Pop an NV off the stack.

`POPi`

Pop an IV off the stack.

`TOPs`

Return the top SV on the stack, but do not pop it. (The macros `TOPpx`, `TOPn`, etc. are analogous)

`TOPm1s`

Return the penultimate SV on the stack. (There is no `TOPm1px`, etc.)

`PUSHs`

Push the scalar onto the stack; you must ensure that the stack has enough space to accommodate it.

`PUSHn`

Set the NV of the target to the given value, and push it onto the stack. `PUSHi`, etc. are analogous.

There is also an `XPUSHs`, `XPUSHn`, etc. which extends the stack if necessary.

`SETs`

This sets the top element of the stack to the given SV. `SETn`, etc. are analogous.

`dTOPss`, `dPOPss`

These declare a variable called `sv`, and either return the top entry from the stack or pop an entry and set `sv` to it.

`dTOPnv`, `dPOPnv`

These are similar, but declare a variable called `value` of the appropriate type. `dTOPiv` and so on are analogous.

In some cases, the PP code is purely concerned with rearranging the stack, and the PP function will call out to another function in `doop.c` to actually perform the relevant operation.

11.5.3. The opcode table and `opcodes.pl`

The header files for the opcode tables are generated from a Perl program called `opcode.pl`. Here is a sample entry for an op:

```
index             index                     ck_index isT@     S S S?
```

The entry is in five columns.

The first column is the internal name of the operator. When `opcode.pl` is run, it will create an enum including the symbol `OP_INDEX`.

The second column is the English description of the operator which will be printed during error messages.

The third column is the name of the "check" function which will be used to optimize this tree; see .

Then come additional flags plus a character which specifies the "flavour" of the op: in this case, `index` is a list op, since it can take more than two parameters, so it has the symbol `@`.

Finally, the "prototype" for the function is given: `S S S?` translates to the Perl prototype `$$;$`, which is indeed the prototype for `CORE::index`.

While most people will never need to edit the op table, it is as well to understand how Perl "knows" what the ops look like. There is a full description of the format of the table, including details of the meanings of the flags, in `opcodes.pl`.

11.5.4. Scatchpads and Targets

PP code is the guts of Perl execution, and hence is highly optimized for speed. One thing that you don't want to do in time-critical areas is create and destroy SVs, because allocating and freeing memory is a slow process. So Perl allocates for each op a *target* SV which is created at compile time. We've seen above that PP code gets the target and uses the `PUSH` macros to push the target onto the stack.

Targets live on the scratchpad, just like lexical variables. `op_targ` for an op is an offset in the current pad; it is the element number in the pad's array which stores the SV that should be used as the target. Perl arranges that ops can reuse the same target if they are not going to collide on the stack; similarly, it also directly uses lexical variables on the pad as targets if appropriate instead of going through a `padsv` operation to extract them. (This is a standard compiler technique called "binding".)

You can tell if an SV is a target by its flags: targets (also known as temporaries) have the `TEMP` flag set, and SVs bound to lexical variables on the pad have the `PADMY` flag set.

11.5.5. The Optimizer

Between compiling the op tree and executing it, Perl goes through three stages of optimization.

The first stage actually happens as the tree is being constructed. Once Perl creates an op, it passes it off to a check routine. We saw above how the check routines are assigned to operators in the op table; an

`index` op will be passed to `ck_index`. This routine may manipulate the op in any way it pleases, including freeing it, replacing it with a different op, or adding new ops above or below it. They are sometimes called in a chain: for instance, the check routine for `index` simply tests to see if the string being sought is a constant, and if so, performs a Fast Boyer-Moore string compilation to speed up the search at runtime; then it calls the general function-checking routine `ck_fun`.

Secondly, the constant folding routine `fold_constants` is called if appropriate. This tests to see whether all of the descendents of the op are constants, and if they are, runs the operator as if it was a little program, collects the result and replaces the op with a constant op reflecting that result. You can tell if constants have been folded by using the "deparse" compiler backend (see Section 11.7.2.3):

```
 % perl -MO=Deparse -e 'print (3+5+8+$foo)'
print 16 + $foo;
```

Here, the `3+5` has been constant-folded into `8`, and then `8+8` is constant-folded to 16.

Finally, the peephole optimizer `peep` is called. This examines each op in the tree in execution order, and attempts to determine "local" optimizations by "thinking ahead" one or two ops and seeing if multiple operations can be combined into one. It also checks for lexical issues such as the effect of `use strict` on bareword constants.

11.5.6. Summary

Perl's fundamental operations are represented by a series of structures, analogous to the structures which make up Perl's internal values. These ops are threaded together in two ways - firstly, into an op tree during the parsing process, where each op dominates its arguments, and secondly, by a thread of execution which establishes the order in which Perl has to run the ops.

To run the ops, Perl uses the code in `pp*.c`, which is particularly macro-heavy. Most of the macros are concerned with manipulating the argument stack, which is the means by which Perl passes data between operations.

Once the op tree is constructed, there are a number of means by which it is optimized - check routines and constant folding which takes place after each op is created, and a peephole optimizer which performs a "dry run" over the execution order.

11.6. Execution

Once we have constructed an op code tree from a program, executing the code is a simple matter of following the chain of `op_next` pointers, and executing the operations specified by each op. The code

which does this is in `run.c`:

```
while ((PL_op = CALL_FPTR(PL_op->op_ppaddr)(aTHX))) {
    PERL_ASYNC_CHECK();
}
```

That's to say, we start with the first op, `PL_op`, and we call the function in its `op_ppaddr` slot. This will return another op, which we assign to `PL_op`, or a null pointer meaning the end of the program. In between executing ops, we perform the "asynchronous check", which despatches signal handlers and other events which may occur between operations.

As we know from looking at XS programming, Perl keeps values between operations on the argument stack. The job of ops is to manipulate the arguments on the stack. For instance, the `add` operator

As you would expect, for most ops, the next operation returned is simply `op_next`, the next one in sequence; in the case of conditional ops, on the other hand, such as logical and, the next op in sequence obviously cannot be determined at compile time, and so these ops return either `op_next` or `op_other` depending on the result of the logical test. For instance, `and` is implemented like this: (in `pp_hot.c`)

```
PP(pp_and)
{
    dSP;
    if (!SvTRUE(TOPs))
        RETURN;
    else {
        --SP;
        RETURNOP(cLOGOP->op_other);
    }
}
```

If the SV on the top of the argument stack does not have a true value, then the `and` cannot be true, so we simply return the next op in the sequence. We don't even need to look at the right hand side of the `and`. If it is true, however, we can discard it by popping the stack and we need to execute the right hand side (stored in `op_other`) to determine whether that is true as well. Hence, we return the chain of operations starting at `op_other`; the `op_next` pointers of these operations will be arranged so as to meet up with the operation after `and`.

* *From netthink*

11.7. The Perl Compiler

We'll finish off our tour of the perl internals by discussing the oft misunderstood Perl compiler.

11.7.1. What is the Perl Compiler?

In 1996, someone

* *(I think it was Chip. Must check.)*

announced a challenge - the first person to write a compiler suite for Perl would win a laptop. Malcolm Beattie stepped up to the challenge, and won the laptop with his `B` suite of modules. Many of these modules have now been brought into the Perl core as standard modules.

The Perl compiler is not just for compiling Perl code to a standalone executable - in fact, some would argue that it's not *at all* for compiling Perl into a standalone executable. We've already seen the use of the `B::Terse` and `B::Tree` modules to help us visualise the Perl op tree, and this should give us a hint as to what the Perl compiler is actually all about.

The compiler comes in three parts: a frontend module, `O`, which does little other than turn on Perl's `-c` (compile only, do not run) flag, and loads up a backend module, such as `B::Terse` which performs a specific compiler task, and the `B` module which acts as a low-level driver.

The `B`, at the heart of the compiler, is a stunningly simple XS module which makes Perl's internal object-like structures - SVs, ops, and so on - into real Perl-space objects. This provides us with a degree of introspection: we can, for instance, write a backend module which traverses the op tree of a compiled program and dump out its state to a file. (This is exactly what the `B::Bytecode` module does.)

It's important to know what the Perl compiler is not. It's not something which will magically make your code go faster, or take up less space, or be more reliable. The backends which generate standalone code generally do exactly the opposite. All the compiler is, essentially, is a way of getting access to the op tree and doing something potentially interesting with it. Let's now take a look at some of the interesting things that can be done with it.

11.7.2. `B::` Modules

There are twelve backend modules to the compiler in the Perl core, and many more besides on CPAN. Here we'll briefly examine those which are particularly helpful to internals hackers or particularly interesting.

11.7.2.1. `B::Concise`

`B::Concise` was written quite recently by Stephen McCamant to provide a generic way of getting concise information about the op tree. It is highly customizable, and can be used to emulate `B::Terse` and `B::Debug`. (see below)

Here's the basic output from `B::Concise`:

```
    % perl -MO=Concise -e 'print $a+$b'
1r <@> leave[t1] vKP/REFC ->(end)
1k    <0> enter ->1l
1l    <;> nextstate(main 7 -e:1) v ->1m
1q    <@> print vK ->1r
1m       <0> pushmark s ->1n
1p       <2> add[t1] sK/2 ->1q
-           <1> ex-rv2sv sK/1 ->1o
1n             <$> gvsv(*a) s ->1o
-           <1> ex-rv2sv sK/1 ->1p
1o             <$> gvsv(*b) s ->1p
```

Each line consists of five main parts:

- a label for this operator (in this case, `1r`)
- a type signifier (`@` is a list operator - think arrays)
- the name of the op and its target, if any, plus any other information about it
- the flags for this operator. Here, `v` signifies void context and `K` shows that this operator has children. The private flags are shown after the slash, and are written out as a longer abbreviation than just one character: `REFC` shows that this op is refcounted.
- finally, the label for the next operator in the tree, if there is one.

Note also that, for instance, ops which have been optimized away to a null are left as "ex-...". The exact meanings of the flags and the op classes are given in the `B::Concise` documentation:

```
=head2 OP flags abbreviations

    v      OPf_WANT_VOID    Want nothing (void context)
    s      OPf_WANT_SCALAR  Want single value (scalar context)
    l      OPf_WANT_LIST    Want list of any length (list context)
    K      OPf_KIDS         There is a firstborn child.
    P      OPf_PARENS       This operator was parenthesized.
                             (Or block needs explicit scope entry.)
    R      OPf_REF          Certified reference.
                             (Return container, not containee).
    M      OPf_MOD          Will modify (lvalue).
    S      OPf_STACKED      Some arg is arriving on the stack.
    *      OPf_SPECIAL      Do something weird for this op (see op.h)

=head2 OP class abbreviations

    0      OP (aka BASEOP)  An OP with no children
    1      UNOP             An OP with one child
    2      BINOP            An OP with two children
    |      LOGOP            A control branch OP
    @      LISTOP           An OP that could have lots of children
```

```
    /      PMOP            An OP with a regular expression
    $      SVOP            An OP with an SV
    "      PVOP            An OP with a string
    {      LOOP            An OP that holds pointers for a loop
    ;      COP             An OP that marks the start of a statement
```

As with many of the debugging `B::` modules, you can use the `-exec` flag to walk the op tree in execution order, following the chain of `op_next`'s from the start of the tree:

```
    % perl -MO=Concise,-exec -e 'print $a+$b'
1k <0> enter
1l <;> nextstate(main 7 -e:1) v
1m <0> pushmark s
1n <$> gvsv(*a) s
1o <$> gvsv(*b) s
1p <2> add[t1] sK/2
1q <@> print vK
1r <@> leave[t1] vKP/REFC
-e syntax OK
```

Amongst other options, (again, see the documentation) `B::Concise` supports a `-tree` option for tree-like ASCII art graphs, and the curious but fun `-linenoise` option.

11.7.2.2. `B::Debug`

`B::Debug` dumps out *all* of the information in the op tree; for anything bigger than a trivial program, this is just way too much information. Hence, to sensibly make use of it, it's a good idea to go through with `B::Terse` or `B::Concise` first, and find which ops you're interested in, and then grep for them.

Some output from `B::Debug` looks like this:

```
LISTOP (0x81121a8)
        op_next         0x0
        op_sibling      0x0
        op_ppaddr       PL_ppaddr[OP_LEAVE]
        op_targ         1
        op_type         178
        op_seq          6433
        op_flags        13
        op_private      64
        op_first        0x81121d0
        op_last         0x8190498
        op_children     3
OP (0x81121d0)
```

```
        op_next         0x81904c0
        op_sibling      0x81904c0
        op_ppaddr       PL_ppaddr[OP_ENTER]
        op_targ         0
        op_type         177
        op_seq          6426
        op_flags        0
        op_private      0
```

As you should know from the ops chapter, this is all the information contained in the op structure: the type of op and its address, the ops related to it, the C function pointer implementing the PP function, the target on the scratchpad this op uses, its type, sequence number, and public and private flags. It also does similar dumps for SVs. You may find the `B::Flags` module useful for "Englishifying" the flags.

11.7.2.3. `B::Deparse`

`B::Deparse` takes a Perl program and turns it into a Perl program. This doesn't sound very impressive, but it actually does so by decompiling the op tree back into Perl. While this has interesting uses for things like serializing subroutines, it's interesting for internals hackers because it shows us how Perl understands certain constructs. For instance, we can see that logical operators and binary "if" are equivalent:

```
%  perl -MO=Deparse -e '$a and do {$b}'
if ($a) {
    do {
        $b;
    };
}
-e syntax OK
```

We can also see, for instance, how the magic that is added by command line switches goes into the op tree:

```
% perl -MO=Deparse -ane 'print'
LINE: while (defined($_ = <ARGV>)) {
    @F = split(" ", $_, 0);
    print $_;
}
-e syntax OK
```

11.7.3. What `B` and `O` Provide

To see how we can built compilers and introspective modules with `B`, we need to see what `B` and the compiler front-end `O` give us. We'll start with `O`, since it's simpler.

11.7.3.1. O

The guts of the O module are very small - only 48 lines of code - because all it intends to do is set up the environment ready for a back-end module. The back-ends are expected to provide a subroutine called compile which processes the options that are passed to it and then returns a subroutine reference which does the actual compilation. O then calls this subroutine reference in a CHECK block.

CHECK blocks were specifically designed for the compiler - they're called after Perl has finished constructing the op tree and before it starts running the code. O calls the B subroutine minus_c which, as its name implies, is equivalent to the command-line -c flag to perl: compile but do not execute the code. It then ensures that any BEGIN blocks are accessible to the back-end modules, and then calls compile from the back-end processor with any options from the command line.

11.7.3.2. B

As we have mentioned, the B module allows Perl-level access to ops and internal variables. There are two key ways to get this access: from the op tree, or from a user-specified Perl "thing".

To get at the op tree, B provides the main_root and main_start functions. These return B::OP-derived objects representing the root of the op tree and the start of the tree in execution order respectively:

```
%  perl -MB -le 'print B::main_root; print B::main_start'
B::LISTOP=SCALAR(0x8104180)
B::OP=SCALAR(0x8104180)
```

For everything else, you can use the svref_2object function which turns some kind of reference into the appropriate B::SV-derived object:

```
%  perl -MB -l
  $a = 5; print B::svref_2object(\$a);
  @a=(1,2,3); print B::svref_2object(\@a);

  B::IV=SCALAR(0x811f9b8)
B::AV=SCALAR(0x811f9b8)
```

(Yes, it's normal that the objects will have the same addresses.)

In this tutorial we'll concentrate on the op-derived classes, because they're the most useful feature of B for compiler construction; the SV classes are a lot simpler and quite analogous.

11.7.4. Using B for Simple Things

OK, so now we have the objects - what can we do with them? B provides accessor methods similar to the fields of the structures in op.h and sv.h. For instance, we can find out the type of the root op like this:

```
$op=B::main_root; print $op->type;
178
```

Oops: op_type is actually an enum, so we can't really get much from looking at that directly; however, B also gives us the name method, which is a little friendlier:

```
$op=B::main_root; print $op->name;
leave
```

We can also use flags, private, targ, and so on - in fact, everything we saw prefixed by op_ in the B::Debug example above.

What about traversing the op tree, then? You should be happy to learn that first, sibling, next and friends return the B::OP object for the related op. That's to say, you can follow the op tree in execution order by doing something like this:

```
#!/usr/bin/perl -cl
use B;
CHECK {
  $op=B::main_start;
  print $op->name while $op=$op->next;
}

print $a+$b;
...
```

Except that's not quite there; when you get to the last op in the sequence, the "enter" at the root of the tree, op_next will be a null pointer. B represents a null pointer by the B::NULL object, which has no methods. This has the handy property that if $op is a B::NULL, then $$op will be zero. So we can print the name of each op in execution order by saying:

```
  $op=B::main_start;
  print $op->name while $op=$op->next and $$op;
```

Walking the tree in normal order is a bit more tricky, since we have to make the right moves appropriate for each type of op: we need to look at both first and last links from binary ops, for instance, but only the first from a unary op. Thankfully, B provides a function which does this all for us:

`walkoptree_slow`. This arranges to call a user-specified method on each op in turn. Of course, to make it useful, we have to define the method...

```
#!/usr/bin/perl -cl
use B;
CHECK {
      B::walkoptree_slow(B::main_root, "print_it", 0);
      sub B::OP::print_it { my $self = shift; print $self->name }
}

print $a+$b;
...
```

Since all ops inherit from `B::OP`, this duly produces:

```
leave
enter
nextstate
print
pushmark
add
null
gvsv
null
gvsv
```

We can also use the knowledge that `walkoptree_slow` passes the recursion level as a parameter to the callback method, and prettify the tree a little, like this:

```
      sub B::OP::print_it {
          my ($self,$level)=@_;
          print "    "x$level, $self->name
      }
```

```
leave
    enter
    nextstate
    print
        pushmark
        add
            null
                gvsv
            null
                gvsv
```

See how we're starting to approximate `B::Terse`? Actually, `B::Terse` uses the `B::peekop` function, a little like this:

```
      sub B::OP::print_it {
          my ($self,$level)=@_;
          print "    "x$level, B::peekop($self);
      }
```

```
LISTOP (0x81142c8) leave
    OP (0x81142f0) enter
    COP (0x8114288) nextstate
    LISTOP (0x8114240) print
        OP (0x8114268) pushmark
        BINOP (0x811d920) add
            UNOP (0x8115840) null
                SVOP (0x8143158) gvsv
            UNOP (0x811d900) null
                SVOP (0x8115860) gvsv
```

All that's missing is that `B::Terse` provides slightly more information based on each different type of op, and that can be easily done by putting methods in the individual op classes: `B::LISTOP`, `B::UNOP` and so on.

Let's finish off our little compiler - let's call it `B::Simple` - by turning it into a module that can be used from the `O` front-end. This is easy enough to do in our case, once we remember that `compile` has to return a callback subroutine reference:

```
package B::Simple;
use B qw(main_root peekop walkoptree_slow);

sub B::OP::print_it {
    my ($self,$level)=@_;
    print "    "x$level, peekop($self);
}

sub compile {
    return sub { walkoptree_slow(main_root, "print_it", 0); }
}

1;
```

If we save the above code as `B/Simple.pm`, we can run it on our own programs with **`perl -MO=Simple ...`**. We have a backend compiler module!

11.7.5. Summary

In this section, we've examined the basics of the Perl compiler: its front-end `O`, the nuts-and-bolts module `B`, and how to write both backend modules using these. Writing compiler modules is really an

excellent way to learn about how the Perl op tree fits together and what the operations signify, so you are encouraged to complete at least some of the following exercises.

Notes

1. Sometimes "Backhaus Normal Form"
2. This is slightly disingenous, as parsing is always done from left to right, but this simplification is easier than getting into the details of how Yacc grammars recognise the precendence of operators.

perl">]>

Chapter 12. Hacking Perl

Perl, just like any other piece of software, is not a finished product; Perl is still being developed, and has a lively development community. Both of the authors are regular contributors to Perl, and we'd like to encourage you to get think about getting involved with Perl's continued maintainance and development. This chapter will tell you what you need to know to start.

12.1. The Development Process

12.1.1. Perl Versioning

Perl has two types of version number: versions before 5.6.0 used a number of the form `x.yyyy_zz`; `x` was the major version number, (Perl 4, Perl 5) `y` was the minor release number, and `z` was the patchlevel. Major releases represented, for instance, either a complete rewrite or a major upheaval of the internals; minor releases sometimes added non-essential functionality, and releases changing the patchlevel were primarily to fix bugs. Releases where `z` was 50 or more were unstable, developers' releases working towards the next minor release.

Now, since, 5.6.0, Perl uses the more standard open source version numbering system - version numbers are of the form `x.y.z`; releases where `y` is even are stable releases, and releases where it is odd are part of the *development track*.

12.1.2. The Development Tracks

Perl development has four major aims: extending portability, fixing bugs, optimizations, and adding language features. Patches to Perl are usually made against the latest copy of the development release; the very latest copy, stored in the Perl repository (see Section 12.1.5 below) is usually called 'bleadperl'.

The bleadperl eventually becomes the new minor release, but patches are also picked up by the maintainer of the stable release for inclusion. While there are no hard and fast rules, and everything is left to the discretion of the maintainer, in general, patches which are bug fixes or address portability concerns (which include taking advantage of new features in some platforms, such as large file support or 64 bit integers) are merged into the stable release as well, whereas new language features tend to be left until the next minor release. Optimizations may or may not be included, depending on their impact on the source.

12.1.3. Perl 5 Porters

All Perl development goes on on the `perl5-porters` mailing list; if you are planning to get involved, a subscription to this is essential.

You can subscribe by sending an email to `perl5-porters-subscribe@perl.org`; you'll be asked to send an email to confirm, and then you should start receiving mail from the list. To send mail, to the list, address the mail to `perl5-porters@perl.org`; you don't have to be subscribed to post, and the list is not moderated. If, for whatever reason, you decide to unsubscribe, simply mail `perl5-porters-unsubscribe@perl.org`.

The list usually receives between 200 and 400 mails a week. If this is too much for you, you can subscribe instead to a daily digest service by mailing `perl5-porters-digest-subscribe@perl.org`. Alternatively, Simon writes a weekly summary of the list, published on the Perl home page (http://www.perl.com/).

There is also a `perl5-porters` FAQ (http://simon-cozens.org/writings/p5p.faq) which explains a lot of this, plus more about how to behave on P5P and how to submit patches to Perl.

12.1.4. Pumpkins and Pumpkings

Development is very loosely organised around the release managers of the stable and the development tracks; these are the two "pumpkings".

Perl development can also be divided up into several smaller sub-systems: the regular expression engine, the configuration process, the documentation, and so on. Responsibility for each of these areas is known as a "pumpkin", and hence those who semi-officially take responsibility for are called "pumpkings".

You're probably wondering why the silly names. It stems from the days before Perl was kept under version control, and people had to manually 'check out' a chunk of the Perl source to avoid conflicts by announcing their intentions to the mailing list; while they were discussing what this should be called, one of Chip Salzenburg's co-workers told him about a system they had used for preventing two people using a tape drive at once: there was a stuffed pumpkin in the office, and nobody could use the drive unless they had the pumpkin.

12.1.5. The Perl Repository

Now Perl is kept in a version control system called Perforce (http://www.perforce.com/), which is hosted by ActiveState, Inc. There is no public access to the system itself, but various methods have been devised to allow developers near-realtime access.

Firstly, there is the Archive of Perl Changes. (ftp://ftp.linux.activestate.com/pub/staff/gsar/APC/) This FTP site contains both the current state of all the maintained Perl versions, and also a directory of changes made to the repository.

Since it's a little inconvenient to keep up to date using FTP, the directories are also available via the software synchronisation protocol rsync (http://rsync.samba.org/). If you have **rsync** installed, you can synchronise your working directory with the bleeding-edge Perl tree (usually called 'bleadperl') in the repository by issuing the command

```
% rsync -avz rsync://ftp.linux.activestate.com/perl-current/ .
```

* *Should also mention the `--delete` should be used periodically to clean out files that are no longer required. Witness the confusion when `t/qu.t` was removed from the distribution.- TJ*

There are also periodic snapshots of bleadperl released by the development pumpking, particularly when some important change happens. These are usually available from a variety of URLs, and always from ftp://ftp.funet.fi/pub/languages/perl/snap/.

12.2. Debugging Aids

There are a number of tools available to developers to help them find and examine bugs in Perl; these tools are, of course, also useful to those creating XS extensions and applications with embedded Perl. There are four major categories: Perl modules such as `Devel::Peek` which allow us to get information about Perl's operation, `perl`'s own debugging mode, convenience functions built into `perl` that we can call to get debugging information, and external applications.

12.2.1. Debugging Modules

We've already seen in Chapter 3 how the `Devel::Peek` module can dump information about SVs; we've also seen the `B::Terse` module for dumping the op tree. The op tree diagrams in the previous chapter were produced using the CPAN module `B::Tree`. There are other modules which we can use to help us get similar information

12.2.1.1. The compiler modules

Due to the way the compiler works, we can use it to get at a lot of information about the op tree. The most extensive information can be found using the `B::Debug` module, which dumps all the fields of all OPs and SVs in the op tree.

`B::Deparse` is useful for understanding the tokeniser - it attempts to turn the op tree back into usable Perl code. For instance, we can see how `perl` implements the **`-p`** switch by running the following code:

```
% perl -MO=Deparse -pe 's/2000/2001/g'
LINE: while (defined($_ = <ARGV>)) {
    s/2000/2001/g;
}
continue {
    print $_;
}
```

Another useful module is `B::Graph`, which produces the same information as `B::Debug`, but does so in the form of a graph.

12.2.1.2. Other Modules

The core module `re` has a debugging mode, **`use re 'debug';`**, which traces the execution of regular expressions. We can use this, for instance, to examine the regular expression engine's backtracking behaviour:

```
% perl -e 'use re "debug"; "aaa" =~/\w+\d/;'
Compiling REx '\w+\d'
size 4 first at 2
   1: PLUS(3)
   2:   ALNUM(0)
   3: DIGIT(4)
   4: END(0)
stclass 'ALNUM' plus minlen 2
Matching REx '\w+\d' against 'aaa'
  Setting an EVAL scope, savestack=3
   0 <> <aaa>             |  1:  PLUS
                           ALNUM can match 3 times out of 32767...
  Setting an EVAL scope, savestack=3
   3 <aaa> <>             |  3:    DIGIT
                              failed...
   2 <aa> <a>             |  3:    DIGIT
                              failed...
   1 <a> <aa>             |  3:    DIGIT
                              failed...
                            failed...
Freeing REx: '\w+\d'
```

Turning to CPAN, the `Devel::Leak` module can be used to detect and trace memory leaks in `perl`.

12.2.2. The Built-in Debugger : `perl -D`

If you configure Perl passing the flag **`-Doptimize='-g'`** to `Configure`, it will do two things - first, it will tell the C compiler to add special debugging information to the object files it produces, and we'll see how that's used in a moment, but it will also define the preprocessor macro `DEBUGGING`, which turns on some special debugging options.

> **Note:** If you're running `Configure` manually, you can turn on debugging in the following way:

```
By default, perl5 compiles with the -O flag to use the optimizer.
Alternately, you might want to use the symbolic debugger, which uses
the -g flag (on traditional Unix systems).  Either flag can be
specified here.  To use neither flag, specify the word "none".

What optimizer/debugger flag should be used? [-O2] -g
```

This allows us to use the **`-D`** flag on the `perl` command line to select the level of debugging we require. The most useful debugging options are as follows:

12.2.2.1. `-Ds`

This turns on stack snapshots, printing a summary of what's on the argument stack each time an operation is performed; this is not *too* useful on its own, but is highly recommended when combined with the **`-Dt`** switch. Here we can see how Perl builds up lists by putting successive values onto the stack, and performs array assignment:

```
% perl -Ds -e '@a = (1,2,3)'

EXECUTING...

    =>
    =>
    =>
    =>  *
    =>  *  IV(1)                                    ❶
    =>  *  IV(1)  IV(2)
    =>  *  IV(1)  IV(2)  IV(3)
    =>  *  IV(1)  IV(2)  IV(3)  *                   ❷
    =>  *  IV(1)  IV(2)  IV(3)  *  GV()             ❸
    =>  *  IV(1)  IV(2)  IV(3)  *  AV()             ❹
    =>                                              ❺
```

❶ Perl pushes each of the values of the list onto the argument stack. The asterisk before the list represents an entry in the mark stack.

❷ Once the list has been built up, Perl places another mark between the right hand side of an assignment and the left hand side, so it knows how many elements are due for assignment.

❸ The array is first placed on the stack as a glob, an entry into the symbol table.

❹ The `rv2av` operator resolves the glob into an AV.

❺ Finally, once the assignment has been made, everything from the first mark is popped off the stack.

12.2.2.2. `-Dt`

This option traces each individual op as it is executed. Let's see the code above again, but this time with a listing of the ops:

```
% perl -Dst -e '@a = (1,2,3)'

EXECUTING...

    =>
(-e:0)  enter
    =>
(-e:0)  nextstate
    =>
(-e:1)  pushmark
    =>  *
(-e:1)  const(IV(1))
    =>  *  IV(1)
(-e:1)  const(IV(2))
    =>  *  IV(1)  IV(2)
(-e:1)  const(IV(3))
    =>  *  IV(1)  IV(2)  IV(3)
(-e:1)  pushmark
    =>  *  IV(1)  IV(2)  IV(3)  *
(-e:1)  gv(main::a)
    =>  *  IV(1)  IV(2)  IV(3)  *  GV()
(-e:1)  rv2av
    =>  *  IV(1)  IV(2)  IV(3)  *  AV()
(-e:1)  aassign
    =>
(-e:1)  leave
```

12.2.2.3. `-Dr`

The **`-Dr`** flag is exactly identical to the **`use re 'debug';`** module discussed above.

12.2.2.4. `-Dl`

This option reports when `perl` reaches an `ENTER` or `LEAVE` statement, and reports on which line and in which file the statement occurred.

12.2.2.5. `-Dx`

This is roughly equivalent to `B::Terse` - it produces a dump of the op tree using the `op_dump` function described below. It's a handy compromise between `B::Terse` and `B::Debug`.

12.2.2.6. `-Do`

This turns on reporting of method resolution: that is, what happens when Perl calls a method on an object or class; it tells you when, for instance, DESTROY methods are called, as well as what happens during inheritance lookups.

12.2.3. Debugging Functions

In addition to this, the Perl core itself defines a number of functions to aid debugging the internal goings-on. These can either be called from debugging sections of your own code, or from a source level debugger. (see below)

12.2.3.1. `sv_dump`

```
void sv_dump(SV* sv);
```

This is roughly equivalent to the `Devel::Peek` module - it allows you to inspect any of Perl's data types. The principle differences between this and `Devel::Peek` is that it is not recursive - for instance, a reference will be dumped like this:

```
SV = RV(0x814fd10) at 0x814ec80
  REFCNT = 1
  FLAGS = (ROK)
  RV = 0x814ec5c
```

and its referent is not automatically dumped. However, it does allow you to get at values that are not attached to a variable, such as arrays and scalars used to hold data internal to `perl`.

12.2.3.2. op_dump

```
void op_dump(OP* op);
```

The **-Dx** debugging option is implemented, essentially, by calling `op_dump(PL_mainroot)`. It takes an op, and lists the op's type, flags, important additional fields and recursively calls itself on the op's children.

12.2.3.3. dump_sub

```
void dump_sub(GV* gv);
```

This extracts the CV from a glob and runs `op_dump` on the root of its op tree.

12.2.4. External Debuggers

There's another way to debug your code, which is usually more useful when you're fiddling around in C. A *source level debugger* allows you to step through your C code line by line or function by function, and execute C code on the fly, just like you'd do with the built-in Perl debugger.

Source level debuggers come in many shapes and sizes: if you're working in a graphical environment such as Microsoft Visual Studio, you may find that there's a debugging mode built into it. Just like with compilers, there are also command-line versions, and we're going to look at another free tool, the GNU Debugger, (**gdb**) although much of what we say will be more or less applicable to other similar debuggers, such as DDD.

12.2.4.1. Compiling for debugging

Unfortunately, before you can use the debugger on a C program, you must compile it with special options. As we've seen above, the debugging option (usually `-g` on command-line compilers) embeds information into the binary detailing the file name and line number for each operation, so that the debugger can, for instance, stop at a specific line in a C source file.

So, before using the debugger, you must recompile Perl with the **`-Doptimize='-g'`** option to `Configure`, as shown in Section 12.2.2

12.2.4.2. Invoking the debugger

* *I would recommend that **ddd** is mentioned somewhere in this chapter. It is a very nice GPL'ed GUI debugger that can debug perl and C. It also can plot arrays. It is layered on top of gdb and is the best free GUI debugger I have seen. - TJ*

We'll assume you're using **gdb**, and you've compiled Perl with the `-g` flag. If you type **`gdb perl`** in the directory in which you built Perl, you should see the following:

```
%  gdb perl
GNU gdb 5.0
Copyright 2000 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "i686-pc-linux-gnu"...
(gdb)
```

If, however, you see the words `(no debugging symbols found)`, you're either in the wrong place or you didn't compile Perl with debugging support.

You can type **`help`** at any time to get a summary of the commands, or **`quit`** (or just press **Ctrl-D**) to leave the debugger.

You can run `perl` without any intervention from the debugger by simply typing **`run`**; this is equivalent to executing `perl` with no command line options, and means that it will take a program from standard input.

To pass command line options to `perl`, put them after the **`run`** command, like this:

```
(gdb) run -Ilib -MDevel::Peek -e '$a="X"; $a++; Dump($a)'
Starting program: /home/simon/patchbay/perl/perl -Ilib -MDevel::Peek -e '$a="X"; $a++; Dump($a)'
SV = PV(0x8146fdc) at 0x8150a18
  REFCNT = 1
  FLAGS = (POK,pPOK)
  PV = 0x8154620 "Y"\0
  CUR = 1
  LEN = 2

Program exited normally.
```

12.2.4.3. Setting breakpoints

Running through a program normally isn't very exciting, though. The most important thing you'll want to do is to choose a place to freeze execution of the program, so that you can examine further what's going on at that point.

The **break** command is used to set a breakpoint, a point in the program at which the debugger will halt execution and bring us back to the `(gdb)` prompt.

break can be given either the name of a function, or a location in the source code of the form ***filename.c*:*lineno***. For instance, in the version of Perl installed here, the main op despatch code is at **run.c:53**:

```
(gdb) break run.c:53
Breakpoint 1 at 0x80ba331: file run.c, line 53.
```

This sets breakpoint number 1, which will be triggered when execution gets to line 53 of `run.c`.

> **Setting breakpoints:** Blank lines, or lines containing comments or preprocessor directives will never get executed, but if you do set a breakpoint on them, the debugger should stop at the next line containing code. This also applies for sections of code which are `#ifdef`'d out.
>
> If you give **break** a function name, be sure to give the name in the `Perl_` namespace: that is, `Perl_runops_debug` instead of `runops_debug`.

Now when you use **run**, execution will halt when it gets to the specified place, and **gdb** will display the number of the breakpoint that was triggered, and the line of code in question for you:

```
(gdb) run -e1
Starting program: /home/simon/patchbay/perl/perl -e1

Breakpoint 1, Perl_runops_debug () at run.c:53
53     } while ((PL_op = CALL_FPTR(PL_op->op_ppaddr)(aTHX)));
```

You can now use the backtrace command, **bt**, to examine the call stack and find out how you got there: (**where** is also available as a synonym for **bt**)

```
(gdb) bt
#0  Perl_runops_debug () at run.c:53
#1  0x805dc9f in S_run_body (oldscope=1) at perl.c:1458
#2  0x805d871 in perl_run (my_perl=0x8146b98) at perl.c:1380
#3  0x805a4d5 in main (argc=2, argv=0xbffff8cc, env=0xbffff8d8)
    at perlmain.c:52
#4  0x40076dcc in __libc_start_main () from /lib/libc.so.6
```

This tells us that we're currently in `Perl_runops_debug`, after being called by `S_run_body` on line 1380 of `perl.c`. **gdb** also displays the value of the arguments to each function, although many of them (those given as hexadecimal numbers) are pointers.

Execution can be restarted by typing **`continue`**; if the code containing a breakpoint is executed again, the debugger will halt once more. If not, the program will run until termination.

You may set multiple breakpoints, simply by issuing more **`break`** commands. If multiple breakpoints are set, the debugger will stop each time execution reaches any of the breakpoints in force.

Unwanted breakpoints can be deleted using the **`delete`** command; on its own, **`delete`** will delete all breakpoints. To delete a given breakpoint, use **`delete`** ***`n`*** where *n* is the number of the breakpoint.

To temporarily turn off a breakpoint, use the **`disable`** and **`enable`** commands.

Good breakpoints to choose when debugging `perl` include the main op despatch code shown above, `main`, `S_parse_body`, `perl_construct`, `perl_destruct` and `Perl_yyparse`. (Not for the faint of heart.)

12.2.4.4. Stepping through a program

While it's perfectly possible to work out the flow of execution just by using breakpoints, it's a lot easier to watch the statements as they get executed. The key commands for this are **`step`**, **`next`**, and **`finish`**.

`step` traces the flow of execution step by step; let's see what happens when we break at the main op despatch look and step through execution:

```
(gdb) run -e1
Starting program: /home/simon/patchbay/perl/perl -e1

Breakpoint 1, Perl_runops_debug () at run.c:53
53     } while ((PL_op = CALL_FPTR(PL_op->op_ppaddr)(aTHX)));
(gdb) step

Perl_pp_enter () at pp_hot.c:1587
1587     djSP;
(gdb) step
1589     I32 gimme = OP_GIMME(PL_op, -1);
(gdb)
1591     if (gimme == -1) {
(gdb)
1592 if (cxstack_ix >= 0)
```

```
(gdb)
1595     gimme = G_SCALAR;
```

> **Tip:** Pressing **Return** repeats the last command.

As we **`step`**ped into the first op, `enter`, **gdb** loaded up `pp_hot.c` and entered `Perl_pp_enter` function. The function in question begins like this:

```
1585  PP(pp_enter)
1586  {
1587      djSP;
1588      register PERL_CONTEXT *cx;
1589      I32 gimme = OP_GIMME(PL_op, -1);
1590
1591      if (gimme == -1) {
1592          if (cxstack_ix >= 0)
1593              gimme = cxstack[cxstack_ix].blk_gimme;
1594          else
1595              gimme = G_SCALAR;
1596      }
1597  ...
```

gdb first stopped at line 1587, which is the first line in the function. The first three lines of the function are, as you might expect, variable definitions. **gdb** does not normally stop on variable definitions, unless they are also assignments. `djSP` happens to be a macro which expands to

```
register SV **sp = PL_stack_sp
```

declaring a local copy of the stack pointer. The next line, however, is not an assignment, which is why **`step`** causes **gdb** to move on to line 1589. **gdb** also skips blank space, so the next line it stops on is 1591.

Since the program enters the `if` statement, we know the `gimme`, the context in which this piece of Perl is being executed, is `-1`, signifying ‘not yet known’. Next we go from the inner `if` statement to the `else` branch, meaning that `cx_stack_ix`, the index into the context stack, is less than zero. Hence `gimme` is set to `G_SCALAR`.

What does this mean in Perl terms? The context stack holds the context for each block; when you call a sub in list context, an entry is popped onto the context stack signifying this. This allows the code which implements `return` to determine which context is expected. Since we are in the outermost block of the program, there are no entries on the context stack at the moment. The code we have just executed sets the context of the outer block to scalar context. (Unfortunately, `wantarray` is only useful inside a subroutine, so the usual way of demonstrating this won’t work. You’ll just have to take our word for it.)

Sometimes **`step`** is too slow-going, and you don't want to descend into a certain function and execute every line in it. For instance, you'll notice after a while that `ENTER` and `SAVETMPS` often appear next to each other and cause `Perl_push_scope` and `Perl_save_int` to be executed. If you're not interested in debugging those functions, you can skip over them using the **`next`** command; they will still be executed, but the debugger will not trace their execution:

```
Breakpoint 2, Perl_pp_enter () at pp_hot.c:1598
1598     ENTER;
(gdb) next
1600     SAVETMPS;
(gdb)
1601     PUSHBLOCK(cx, CXt_BLOCK, SP);
(gdb)
1603     RETURN;
(gdb)
```

Alternatively, you can run the current function to its conclusion without tracing it using the **`finish`** command.

```
    (gdb) step
Perl_runops_debug () at run.c:42
42 PERL_ASYNC_CHECK();
(gdb)
43 if (PL_debug) {
(gdb)
53     } while ((PL_op = CALL_FPTR(PL_op->op_ppaddr)(aTHX)));
(gdb)
Perl_pp_nextstate () at pp_hot.c:37
37     PL_curcop = (COP*)PL_op;
(gdb) finish
Run till exit from #0  Perl_pp_nextstate () at pp_hot.c:37
0x80ba64b in Perl_runops_debug () at run.c:53
53     } while ((PL_op = CALL_FPTR(PL_op->op_ppaddr)(aTHX)));
Value returned is $1 = (OP *) 0x814cb68
```

Here we step over the main op despatch loop until `Perl_pp_nextstate` is called. Since we're not particularly interested in that function, we call **`finish`** to let it run. The debugger then confirms that it's running `Perl_pp_nextstate` until the function exits, displays where it has returned to, and the value returned from the function.

> **Emacs makes it easy:** If you're a user of the **emacs** editor, you might find the `gdb` major mode to be extremely helpful; it automatically opens any source files that **gdb** refers to, and can trace the flow of control in the source buffers, making it very easy for you to see what's going on around the source that's currently being executed.

12.2.4.5. Evaluating expressions

We can now perform most of the debugging we need with ease, but there's one more feature of **gdb** that makes it even easier. The **`print`** command allows you to execute C expressions on the fly and display their results.

There is, unfortunately, one drawback; **gdb** doesn't know anything about pre-processor macros, so you need to expand the macros yourself. For instance, to find the reference count of an SV, you can't say

```
(gdb) print SvREFCNT(sv)
No symbol "SvREFCNT" in current context.
```

Instead, you have to say:

```
(gdb) print sv->sv_refcnt
$1
```

Or even, to look at the contents of the SV,

```
(gdb) print *sv
$2 = {sv_any = 0x8147a10, sv_refcnt = 1, sv_flags = 536870923}
```

You may also use **`print`** to call C functions, such as the debugging functions mentioned above:

```
    (gdb) print Perl_sv_dump(sv)
SV = PV(0x8146d14) at 0x8150824
  REFCNT = 1
  FLAGS = (POK,READONLY,pPOK)
  PV = 0x8151968 "hello"\0
  CUR = 5
  LEN = 6
$9 = void
```

Using these functions in conjunction with the execution-tracing commands of **gdb** should allow you to examine almost every area of Perl's internals.

12.2.4.6. Debugging XS code

There's one little wrinkle, however, when it comes to debugging XS modules. The first problem with XS is that the modules are usually dynamically loaded into memory; that means that when `perl` starts, the functions aren't loaded, and that means that when **gdb** starts, it can't find them.

The solution to this is to choose a breakpoint after the XS module has been dynamically loaded; a good place is `S_run_body` - here the **`BEGIN`** blocks have been processed and hence all **`use`**'d modules have been loaded. This is just before the main part of the script is executed. If this is too late for your

debugging, another good place to stop is inside the dynamic loading module, `DynaLoader`. `XS_DynaLoader_dl_load_file` is called for each module that needs to be dynamically loaded.

> **Note:** Don't forget that to effectively debug an XS module, you'll need to recompile it with the debugging flag, **`-g`**. The "official" way to do this is to run `Makefile.PL` as follows:
>
> ```
> % perl Makefile.PL OPTIMIZE=-g
> ```
>
> However, it's also possible to hack the **`OPTIMIZE=`** line in the `Makefile` itself. (But don't tell anyone I said that.)

The next small problem is that the function names of XS functions are mangled from the names you give them in the `.xs` file. You should look at the `.c` file produced by **xsubpp** to determine the real function name.

For instance, the XS function `sdbm_TIEHASH` in the XS code for the `SDBM_File` becomes `XS_SDBM_File_TIEHASH`.

The rules for this mangling are regular (Section 6.10):

- The **`PREFIX`** given in the XS file is removed from the function name. Hence, `sdbm_` is stripped off to leave `TIEHASH`.
- The **`PACKAGE`** name (**`SDBM_File`**) undergoes "C-ification" (Any package separators, **`::`**, are converted to underscores) and this is added to the beginning of the name: `SDBM_File_TIEHASH`
- Finally, `XS_` is prefixed to the name to give `XS_SDBM_File_TIEHASH`.

12.3. Creating a Patch

12.3.1. How to Solve Problems

There are a few standard design goals that you should hold in mind when considering how to approach a Perl patch; there's also quite a lot of unwritten folklore that explains why certain patches 'feel' better than others. Here is an incomplete list of some of the more important principles that we've picked up over the years.

- The most important rule of all: you may not break old code. Perl 5 can still quite happily run some code that is positively ancient, even dating back to the Perl 1 days; we pride ourselves on backwards compatibility. Hence, nothing that you do should break that compatibility.

This has a few direct implications: adding new syntax is tricky. Adding new operators is pretty much right out - if I wanted to introduce a `chip` operator which took a character off the beginning of a string, that would break any code which defined a `chip` subroutine itself.

- Solve problems as generally as possible - platform specific **`ifdef`**s are frowned upon unless absolutely and obviously necessary. Try to avoid repetition of code. If you've got a good, general routine that can be used in other places of the Perl core, move it out to a separate function and change the rest of the core to use it.

  For instance, there needed to be a way for Perl to perform arbitrary transformations on incoming data - for example, to mark it as UTF8-encoded, or convert it between different character encodings. The intial idea was to extend the source filter mechanism to apply not just to the source file input, but also to any filehandle. However, the more general solution was an extension of the Perl IO abstraction to a 'layered' model where transformation functions could be applied to various layers; then source filters could be re-implemented in terms of this new IO system.

- Change as *little* as possible to get the job done, especially when you're not well-known as a solid porter. Big sweeping changes scare people, whether or not they're correct. It's a lot easier to check a ten-line patch for potential bugs than a hundred-line patch.
- Don't do it in the core unless it needs to be done in the core. If you can do it in a Perl module or an XS module, it's unlikely that you need to do it in the core.

  As an example, DBM capability was moved out of the core into a bunch of XS modules; this also had the advantage that you could switch between different DBM libraries at runtime, and you had the extensibility of the `tie` system that could be used for things other than DBMs.

- Try to avoid introducing restrictions - even on things you haven't thought of yet. Always leave the door open for more interesting work along the same lines.

  A good example of this is lvalue subroutines, which were introduced in Perl 5.6.0 - once you have lvalue subroutines, why not lvalue method calls or even lvalue overloaded operators?

Some of the goals, however, are just things which you have to pick up in time and/or may depend on the outlook of the pumpking and any major work that's going on at the time - for instance, during the reorganisation of the IO system mentioned above, any file handling patches would be carefully scrutinised to make sure they wouldn't have to be rewritten once the new system was in place. Hence, it's not really possible to give hard-and-fast design goals, but if you stick to the above, you won't go far wrong.

12.3.2. Autogenerated files

One thing you need to be careful of is that a number of files should not be patched directly, since they are generated from other (usually Perl) programs.

Most of these files are clearly marked, but the most important one bears repeating: if you add a new function to the core, you must add an entry to the table at the end of `embed.pl`. This ensures a number of things: firstly, that a correct function prototype is generated and placed in `protos.h`; secondly, that any documentation for that function is automatically extracted, and thirdly, that the namespace for the function is automatically handled. (See the note below)

The syntax for entries in the table is explained in the documentation file `perlguts.pod`

> **Note:** Perl's internal functions are carefully named so that when Perl is embedded in another C program, they do not override any functions that the C program defines. Hence, all internal functions should be named `Perl_something`, apart from static functions which are by convention named `S_something`. `embed.h` uses a complicated system of automatically-generated **`#define`**s to allow you to call your function as `something()` inside the core and in XSUBs, but `Perl_something` must be used by embedders.

You must remember to rerun `embed.pl` after doing this. The Make target **`regen_headers`** will call all the Perl programs which generate other files.

A special exception is `perly.c`, which is generated from running **byacc** on `perly.h` and then being fixed up with a patch. In the *extraordinarily* unlikely event that you need to fiddle with the Perl grammar in `perly.y`, you can run the Make target **`run_byacc`** to call **byacc** and then fix up the resulting C file.

* *A footnote might be required to point out that VMS requires a slightly different method of generating perly.c. Wasn't there a special request to cc perl-mvs mailing list when ever perly.c is regenerated?*

For changes which involve autogenerated files, such as adding a function to the core or changing a function's prototype, it's only necessary to provide a patch for the generating program and leave a note to the effect that **`regen_headers`** should be run. You should not include, for instance, a patch to `protos.h`.

12.3.3. The Patch Itself

Patching styles vary, but the recommended style for Perl is a unified diff. If you're changing a small number of files, copy, say, `sv.c` to `sv.c~`, make your changes, and then run:

```
% diff -u sv.c~ sv.c > /tmp/patch
% diff -u sv.h~ sv.h >> /tmp/patch
```

and so on for each file you change.

If you are doing this, remember to run **diff** from the root of the Perl source directory. Hence, if you're patching XS files in `ext/`, you should say:

```
% diff -u ext/Devel/Peek/Peek.xs~ ext/Devel/Peek/Peek.xs >> /tmp/patch
```

For larger patches, you may find it easier to do something like this:

```
/home/me/work % rsync -avz rsync://ftp.linux.activestate.com/perl-current/ bleadperl
/home/me/work % cp -R bleadperl myperl
/home/me/work % cd myperl
/home/me/work/myperl % Make your changes...
/home/me/work/myperl % cd ..
/home/me/work % diff -ruN bleadperl myperl > /tmp/patch
```

This will create a patch which turns the current bleadperl into your personal perl source tree. If you do this, please remember to prune your patch for autogenerated files and also things which do not belong in the source distribution. (Any test data you have used, or messages about binary files.)

> **makepatch:** An alternative tool which may make patching easier is Johan Vroman's **makepatch**, available from `$CPAN/authors/id/JV/`. This automates many of the above steps. Some swear by it, but some of us are stuck in their ways and do things the old way...

12.3.4. Documentation

If you change a feature of Perl which is visible to the user, you must, must, must update the documentation. Patches are not complete if they do not contain documentation.

Remember that if you introduce a new warning or error, you need to document it in `pod/perldiag.pod`.

Perl 5.6.0 introduced a system for providing documentation for internal functions, similar to Java's **javadoc**. This 'apidoc' is extracted by `embed.pl` and ends up in two files: `pod/perlapi.pod` is the file containing documentation for functions which are deemed suitable for XS authors[1], and `pod/perlintern.pod` contains the documentation for all other functions. ('Internal' functions)

* *If `perlapi.pod` is the XS interface shouldn't PUSH[inpu] and XPUSH[inpu] be moved to `perlintern.pod` folowing the recent comment by Sarathay that XS writers should not use them (I added comment to that effect in advxs chapter.*

apidoc is simply POD embedded in C comments; you should be able to pick up how it is used by looking around the various C files. If you add apidoc to a function, you should turn on the `d` flag in that function's `embed.pl` entry.

12.3.5. Testing

The `t/` directory in the Perl source tree contains at great many (294, at last count) regression test scripts which ensure that Perl is behaving as it should. When you change something, you should make sure that your changes have not caused any of the scripts to break - they have been specially designed to try out as many unexpected interactions as possible.

You should also add tests to the suite if you change a feature, so that your changes don't get disturbed by future patching activity. Tests are in the ordinary style used for modules, so remember to update the `"1..n"` line at the top of the test.

12.3.6. Submitting your patch

Once you've put together a patch, which includes documentation and new tests, it's time to submit it to P5P. Your subject line should include the tag `[PATCH]`, with optionally a version number or name, or the name of the file you're patching: `[PATCH bleadperl]`, or `[PATCH sv.c]`. This is to allow the pumpking to easily distinguish possible patches to be integrated from the usual list discussion. You should also put a brief description of what you're solving on the subject line: for instance, `[PATCH blead] Fix B::Terse indentation`.

The body of your mail should be a brief discussion of the problem (just some Perl code which demonstrates the problem is adequate) and how you've solved it. Then insert your patch directly into the body of the mail - try to avoid sending it as an attachment. Also, be careful with cut-and-pasting your patch in, because this may corrupt line wrapping or convert tabs to spaces.

Once you're ready, take a deep breath, and hit send!

12.4. Perl 6 : The Future of Perl

12.4.1. A History

At the Perl Conference in July 2000, Chip Salzenburg called a 'brainstorming session' meeting of some eminent members of the Perl community to discuss the state of Perl. Chip wanted some form of 'Perl Constitution' to resolve some perceived problems in Perl 5 development, but Jon Orwant suggested (in a

particularly vivid and colourful way) that there were deeper problems in the state of Perl and the Perl community that should be fixed by a completely new version of Perl.

The majority consensus was that this was a good idea, and Larry picked up on it. It was then presented to the main perl5-porters meeting the same afternoon, and various people offered to take some roles in the development team. Larry announced the start of Perl 6 development in his keynote 'State of the Onion' address the following day.

There was then a period of feeling around for the best way to organise the development structure of Perl 6; the single-mailing-list model of Perl 5 was prone to infighting, and the pumpking system was problematic as Perl was beginning to get too big for a single person to maintain and cases of 'pumpking burnout' too common.

The concensus was that design should be split between a number of 'working groups', and these would each have a chair. The first two working groups were perl6-language and perl6-internals, for language design proposals and implementation design respectively. The busier working groups spawned sub-groups for discussion of more focused topics, and developers were encouraged to express their desires for language change in formal 'Requests for Changes'.

The comments stage ended on October 1st, 2000, after 361 RFCs were submitted. These went to Larry, who sat down to the gruelling task of reading each one to assess its merits. Larry then responded with the Perl 6 language design on XXXX.

* *And this is where we have to stop because it hasn't happened yet. :)*

12.4.2. Design Goals

12.4.3. What happens next

12.4.4. The future for Perl 5

So if Perl 6 is coming and it's going to be so cool, why have we just written a book about Perl 5? Well, for starters, Perl 6 is going to take quite a while to get completed - writing an interpreter from scratch is quite an ambitious exercise! - and it'll then also take a very long time to become generally accepted.

Perl 5 will continue to be developed, up until the release of version 5.8.0, and even then maintainance will continue throughout the lifespan of Perl 6. Perl 5 is not going to become unsupported.

In short, Perl 5 isn't going away anytime soon. Remember how long it took to get rid of all those Perl 4 interpreters and code everywhere? That was when we *wanted* to get rid of it; now that Perl 6 is likely to be non-compatible with Perl 5, we can expect real uptake to be even slower. Since there's an awful lot of working Perl 5 code out there, people aren't going to want to break it all by upgrading to Perl 6.

12.5. Summary

This chapter looked at how to develop `perl` itself; the development process and the perl5-porters mailing list. As well as looking at some of the tools available to help us develop, such as `perl`'s debugging mode and the GNU Debugger, we also looked at the less technical parts of being a Perl porter - how to approach Perl maintainance, and how to get patches submitted and integrated to the Perl core.

We also looked at Perl 6, and gave a glimpse as to how things are likely to be in the future.

12.6. Related Reading

More thoughts on patching Perl can be found in the perl5-porters FAQ at http://simon-cozens.org/writings/p5p.faq, Simon's So You Want To Be A Perl Porter? (http://simon-cozens.org/writings/perlhacktut.html), and in `pod/perlhack.pod`, `Porting/patching.pod` and `Porting/pumpking.pod` in the Perl distribution.

Notes

1. Chapter 4 of this book was developed by starting from `pod/perlapi.pod`, and, in fact, we contributed back some pieces of chapter 4 as apidoc.