

General munging practices



What this chapter covers:

- Processes for munging data
- Data structure designs
- Encapsulating business rules
- The UNIX filter model
- Writing audit trails

When munging data there are a number of general principles which will be useful across a large number of different tasks. In this chapter we will take a look at some of these techniques.

2.1 *Decouple input, munging, and output processes*

When written in pseudocode, most data munging tasks will look very similar. At the highest level, the pseudocode will look something like this:

```
Read input data
Munge data
Write output data
```

Obviously, each of these three subtasks will need to be broken down into greater detail before any real code can be written; however, looking at the problem from this high level can demonstrate some useful general principles about data munging.

Suppose that we are combining data from several systems into one database. In this case our different data sources may well provide us with data in very different formats, but they all need to be converted into the same format to be passed on to our data sink. Our lives will be made much easier if we can write one output routine that handles writing the output from all of our data inputs. In order for this to be possible, the data structures in which we store our data just before we call the combined output routines will need to be in the same format. This means that the data munging routines need to leave the data in the same format, no matter which of the data sinks we are dealing with. One easy way to ensure this is to use the same data munging routines for each of our data sources. In order for this to be possible, the data structures that are output from the various data input routines must be in the same format. It may be tempting to try to take this a step further and reuse our input routines, but as our data sources can be in completely different formats, this is not likely to be possible. As figures 2.1 and 2.2 show, instead of writing three

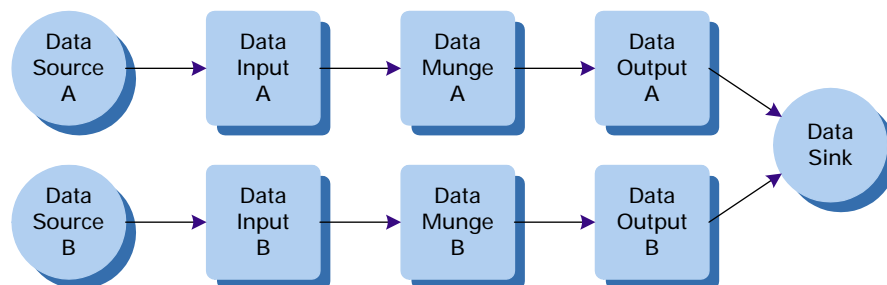


Figure 2.1 Separate munging and output processes

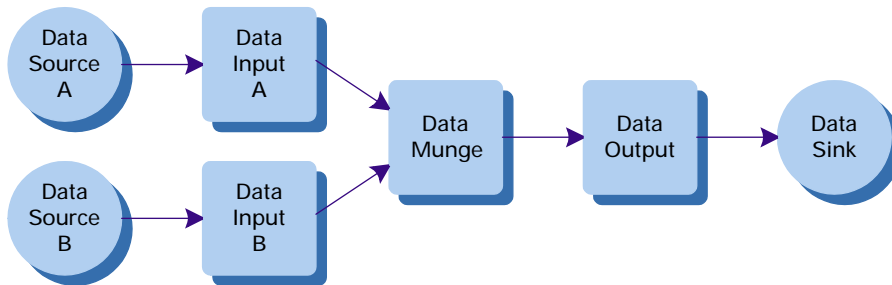


Figure 2.2 Combined munging and output processes

routines for each data source, we now need only write an input routine for each source with common munging and output routines.

A very similar argument can be made if we are taking data from one source and writing it to a number of different data sinks. In this case, only the data output routines need to vary from sink to sink and the input and munging routines can be shared.

There is another advantage to this decoupling of the various stages of the task. If we later need to read data from the same data source, or write data to the same data sink for another task, we already have code that will do the reading or writing for us. Later in this chapter we will look at some ways that Perl helps us to encapsulate these routines in reusable code libraries.

2.2 *Design data structures carefully*

Probably the most important way that you can make your data munging code (or, indeed, any code) more efficient is to design the intermediate data structures carefully. As always in software design, there are compromises to be made, but in this section we will look at some of the factors that you should consider.

2.2.1 *Example: the CD file revisited*

As an example, let's return to the list of compact disks that we discussed in chapter 1. We'll assume that we have a tab-separated text file where the columns are artist, title, record label, and year of release. Before considering what internal data structures we will use, we need to know what sort of output data we will be creating. Suppose that we needed to create a list of years, together with the number of CDs released in that year.

Solution 1: simple hash

The immediately obvious solution is to use a hash in which the keys are years and the values are the numbers of CDs. In this case, there will be no need for a separate data munging process, as all of the required munging will be carried out in the input routine. We might create a first draft script something like this:

```
my %years;
while (<STDIN>) {
    chomp;
    my $year = (split /\t/)[3];

    $years{$year}++;
}

foreach (sort keys %years) {
    print "In $_, $years{$_} CDs were released.\n";
}
```

1971	1
1987	1
1993	1
1996	1
1997	1
1998	1

Figure 2.3 Initial data structure design

This provides a solution to our problem in a reasonably efficient manner. The data structure that we build is very simple and is shown in figure 2.3.

Solution 2: adding flexibility

But how often are requirements as fixed as these?¹ Suppose later someone decides that, instead of having a list of the number of CDs released, they also need a list of the actual CDs. In this case, we will need to go back and rewrite our script completely to something like this:

```
my %years;
while (<STDIN>) {
    chomp;
    my ($artist, $title, $label, $year) = split /\t/;

    my $rec = {artist => $artist,
               title => $title,
               label => $label};
    push @ {$year{$year}}, $rec;
}

foreach my $year (sort keys %years) {
    my $count = scalar @ {$years{$year}};
    print "In $year, $count CDs were released.\n";
    print "They were:\n";
    print map { "$_->{title} by $_->{artist}\n" } @ {$years{$year}};
}
```

¹ There are, of course, many times when the requirements won't change—because this is a one-off data load process or you are proving a concept or building a prototype.

As you can see, this change has entailed an almost complete rewrite of the script. In the new version, we still have a hash where the keys are the years, but each value is now a reference to an array. Each element of this array is a reference to a hash which contains the artist, title, and label of the CD. The output section has also grown more complex as it needs to extract more information from the hash.

Notice that the hash stores the CD's label even though we don't use it in the output from the script. Although the label isn't required in our current version, it is quite possible that it will become necessary to add it to the output at some point in the future. If this happens we will no longer need to make any changes to the input section of our script as we already have the data available in our hash. This is, in itself, an important data munging principle—if you're reading in a data item, you may as well store it in your data structure. This can be described more succinctly as “Don't throw anything away.” This improved data structure is shown in figure 2.4.

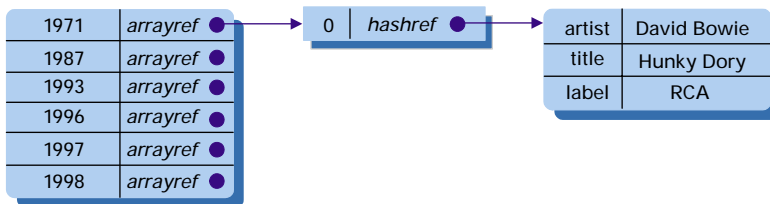


Figure 2.4 Improved data structure design

Solution 3: separating parsing from munging

What happens, however, if the requirements change completely so that we now need to display counts by artist for a different report? Our current script is of no use at all. There is no part of it that is reusable to help us achieve our new goals. Perhaps we need to rethink our strategy from the start.

In all of the scripts above we were not following the advice of the previous section. We were trying to do too much in the input section and left ourselves nothing to do in the data munging section. Perhaps if we went back to a more decoupled approach, we would have more success.

This leaves us contemplating our original question again—what structure would offer the best way to represent our data inside the program? Let's take another look at the data. What we have is a list of records, each of which has a well-defined set of attributes. We could use either a hash or an array to model our list of records and we have the same choices to model each individual record. In this case we will use an

array of hashes² to model our data. A good argument could be made for just about any other combination of arrays and hashes, but the representation that I have chosen seems more natural to me. Our input routine will therefore look like this:

```
my @CDs;
sub input {
  my @attrs = qw(artist title label year);
  while (<STDIN>) {
    chomp;
    my %rec;
    @rec{@attrs} = split /\t/;
    push @CDs, \%rec;
  }
}
```

This third and final data structure is shown in figure 2.5.

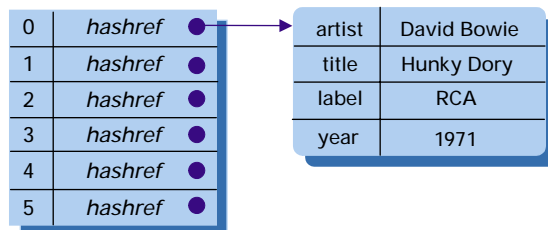


Figure 2.5
Final data structure

More examples: using our flexible data structure

Based on this data structure, we can then write any number of data munging routines to produce specific output reports. For example, to produce our original list of the CDs released in a year we would write a routine like this:

```
sub count_cds_by_year {
  my %years;

  foreach (@CDs) {
    $years{$_->{year}}++;
  }

  return \%years;
}
```

This routine returns a reference to a hash which is identical in structure to our original hash and can therefore be printed out using code identical to the output section of our original script.

² Or, more accurately, an array of references to hashes.

To produce a list of the number of CDs released by each artist we can write a similar routine like this:

```
sub count_cds_by_artist {
    my %artists;

    foreach (@CDs) {
        $artists{$_->{artist}}++;
    }

    return \%artists;
}
```

In fact these two routines are so similar, that it is possible to write a generic version which handles both of these cases (along with the cases where you want to count CDs by label or even by title).

```
sub count_cds_by_attr {
    my $attr = shift;

    my %counts;

    foreach (@CDs) {
        $counts{$_->{$attr}}++;
    }

    return \%counts;
}
```

A complete program to produce counts of CDs by any attribute which is passed in on the command line would look like this:

```
#!/usr/bin/perl -w

use strict;

my @CDs;

sub input {
    my @attrs = qw(artist title label year);
    while (<STDIN>) {
        chomp;
        my %rec;
        @rec{@attrs} = split /\t/;
        push @CDs, \%rec;
    }
}

sub count_cds_by_attr {
    my $attr = shift;

    my %counts;

    foreach (@CDs) {
        $counts{$_->{$attr}}++;
    }
}
```

```
    }
    return \%counts;
}

sub output {
    my $counts = shift;
    foreach (sort keys %{$counts}) {
        print "$_: $counts->{$_}\n";
    }
}

my $attr = shift;

input();
my $counts = count_cds_by_attr($attr);
output($counts);
```

And, assuming that the program file is called `count_cds.pl` and you have the CD list in a file called `cd.txt`, it would be called like this:

```
count_cds.pl year < cds.txt > cds_by_year.txt
count_cds.pl label < cds.txt > cds_by_label.txt
count_cds.pl artist < cds.txt > cds_by_artist.txt
count_cds.pl title < cds.txt > cds_by_title.txt
```

In most cases you will have to make similar decisions when designing data structures. A data structure that is designed for one job will, in general, be simpler than one that is designed to be more flexible. It is up to you to decide whether it is worth taking the extra effort to make the design more flexible. (A hint—it usually is!)

2.3 *Encapsulate business rules*

Much of the logic in your data munging programs will be modeling what might be described as “business rules.” These are the rules about what particular data items mean, what their valid sets of values are, and how they relate to other values in the same or other records.³ Examples of these three types of business rules are:

- Customer number is a unique identifier for a customer.
- Customer number is always in the format CUS-XXXXX, where XXXXX is a unique integer.
- Each customer record must be linked to a valid salesperson record.

³ I’ve described these constraints as “business rules,” as I think that’s easier to remember than something like “domain specific constraints.” Of course, what you’re encoding might well have nothing to do with “business.”

In any system where these data items are used, the business rules must always hold true. No matter what your program does to a customer record, the customer number must remain unique and in the given format, and the customer must be linked to a valid salesperson record. Nothing that you do to a customer record is allowed to leave the data in a state that breaks any of these rules.

2.3.1 *Reasons to encapsulate business rules*

In a real-world system, there will probably be many other programs that are accessing the same data items for their own purposes. Each of these other programs will have to abide by exactly the same set of business rules for each customer record that it accesses. Therefore each of these programs will have logic within it that encodes these rules. This can lead to a couple of problems:

- It is possible that not every programmer who writes these programs has exactly the same understanding of the rules. Therefore, each program may have subtly different interpretations of the rules.
- At some point in the future these rules may be changed. When this happens, the same changes in logic will need to be made to each of the programs that use the existing business rules. This may be a large job, and the more times the changes have to be made, the higher the chance that errors will creep in.

2.3.2 *Ways to encapsulate business rules*

The most common solution to both of these problems is to write code that models the business rules in one place and to reuse that code in each program that needs to use the rules. Most programming languages have a facility that allows code to be reused across many programs, and Perl has a particularly powerful implementation of this functionality.

In Perl you would create a module that contains the business rules for a particular type of business record (say, a customer) and include this module in any other Perl programs that needed to understand the business rules that control the use of customer records. In fact, Perl gives you a couple of ways to implement this functionality. If your rules are relatively simple you can write a module that contains functions called things like `get_next_custno` or `save_cust_record` which get called at relevant points in your programs. For a more robust solution, you should consider writing a Perl object to implement your customer record. Let's look at examples of both of these approaches.

2.3.3 Simple module

Assume that we want to model the three business rules mentioned at the start of this section. We will write a module called `Customer_Rules.pm` that will contain the two functions `get_next_cust_no` and `save_cust_record` which we suggested above. The following example omits some of the lower level functions.

```
package Customer_Rules;

use strict;
use Carp;
use vars qw(@EXPORT @ISA);

@EXPORT = qw(get_next_cust_no save_cust_record);
@ISA = qw(Exporter);

require Exporter;

sub get_next_cust_no {
    my $prev_cust = get_max_cust_no()
        || croak "Can't allocate new customer reference.\n";

    my ($prev_no) = $prev_cust =~ /(\d+)/;
    $prev_no++;

    return "CUS-$prev_no";
}

sub save_cust_record {
    my $cust = shift;

    $cust->{cust_no} ||= get_next_cust_no();

    is_valid_sales_ref($cust->{salesperson})
        || croak "Invalid salesperson ref: $cust->{salesperson}.";

    write_sales_record($cust);
}
```

How Customer_Rules.pm works

In this example we have encapsulated our business rules in functions which, in turn, make use of other lower level functions. These lower level functions haven't been shown, as they would contain implementation-specific details which would only cloud the picture.

In the `get_next_cust_no` function we begin by getting the customer number of the most recently created customer record. This might be stored in a database table or in a text file or in some other format. In all of these cases there will need to be some kind of transaction-level locking to ensure that no other process gets the same value for the previous customer number. This would potentially lead to nonunique customer numbers in the system, which would break one of our business rules.

Having retrieved the previous customer number we simply extract the integer portion, increment it, and return it with the string `CUS-` prepended to it.

In the `save_cust_record` function, we assume that the customer record is stored internally in some complex data structure and that we are passed a reference to that structure. The first thing that we do is to ensure that we have a customer number in the record. We then check that the `$cust->{salesperson}` value represents a valid salesperson in our system. Again, the list of valid salespeople could be stored in a number of different forms. It may be possible that more data is required in order to validate the salesperson code. For example, a salesperson may only deal with customers in a certain region. In this case the region in which the customer is based will also need to be passed into the `is_valid_sales_ref` function.

Eventually, we get a true or false value back from `is_valid_sales_ref` and can proceed appropriately. If the salesperson is valid, we can write the customer record to whatever storage medium we are using; otherwise, we alert the user to the error. In a real-world system many other similar checks would probably need to be carried out.

Using Customer_Rules.pm

Having produced this module, we can make it available to all programmers who are writing applications by putting it into a project-wide library directory. To make use of these functions, a programmer only needs to include the line:

```
use Customer_Rules;
```

in a program. The program will now have access to the `get_next_cust_no` and `save_cust_record` functions. Therefore, we can ensure that every program has the same interpretation of the business rules and, perhaps more importantly, if the business rules change, we only need to change this module in order to change them in each program.

2.3.4 *Object class*

While the module of the previous section is useful, it still has a number of problems; not the least of which is the fact that the structure of the customer record is defined elsewhere in the application. If the module is reused in a number of applications, then each application will define its own customer record and it is possible that the definitions will become out of step with each other. The solution to this problem is to create an object class.

An object defines both the structure of a data record and all of the methods used to operate on the record. It makes the code far easier to reuse and maintain. A full discussion of the advantages of object-oriented programming (OOP) is beyond the scope of this book, but two very good places to get the full story are the *perltoot* manual page and Damian Conway's *Object Oriented Perl* (Manning).

Let's examine a cut-down customer object which is implemented in a module called `Customer.pm`.

```
package Customer;

use strict;

sub new {
    my $thing = shift;
    my $self = {};

    bless $self, ref($thing) || $thing;

    $self->init(@_);
    return $self;
}

sub init {
    my $self = shift;

    # Extract various interesting things from
    # @_ and use them to create a data structure
    # that represents a customer.
}

sub validate {
    my $self = shift;

    # Call a number of methods, each of which validates
    # one data item in the customer record.
    return $self->is_valid_sales_ref
        && $self->is_valid_other_attr
        && $self->is_valid_another_attr;
}

sub save {
    my $self = shift;

    if ($self->validate) {
        $self->{cust_no} ||= $self->get_next_cust_no;
        return $self->write;
    } else {
        return;
    }
}

# Various other object methods are omitted here, for example
# code to retrieve a customer object from the database or
# write a customer object to the database.

1; # Because all modules should return a true value.
```

The advantage that this method has over the previous example is that in addition to modeling the business rules that apply to a customer record, it defines a standard

data structure to store customer data and a well defined set of actions that can be performed on a customer record. The slight downside is that incorporating this module into a program will take a little more work than simply using the functions defined in our previous module.

Example: using *Customer.pm*

As an example of using this module, let's look at a simple script for creating a customer record. We'll prompt the user for the information that we require.

```
use Customer;

my $cust = Customer->new;

print 'Enter new customer name: ';
my $name = <STDIN>;
$cust->name($name);

print 'Enter customer address: ';
my $addr = <STDIN>;
$cust->address($addr);

print 'Enter salesperson code: ';
my $sp_code = <STDIN>;
$cust->salesperson($sp_code);

# Write code similar to that above to get any other
# required data from the user.

if ($cust->save) {
    print "New customer saved successfully.\n";
    print "New customer code is ", $cust->code, "\n";
} else {
    print "Error saving new customer.\n";
}
```

In this case we create an empty customer object by calling the `Customer->new` method without any parameters. We then fill in the various data items in our customer object with data input by the user. Notice that we assume that each customer attribute has an access method which can be used to set or get the attribute value.⁴

⁴ This is a common practice. For example, the `name` method counts the number of parameters that have been sent. If it has received a new value then it sets the customer's name to that value; if not, it just returns the previous value.

An alternative practice is to have two separate methods called `get_name` and `set_name`. Which approach you use is a matter of personal preference. In either case, it is generally accepted that using access methods is better than accessing the attributes directly.

Having filled in all of the required data, we called `$cust->save` to save our new record. If the save is successful, the code attribute will have been filled in and we can display the new customer’s code to the user by way of the `$cust->code` attribute access method.

If, on the other hand, we wanted to access an existing customer record, we would pass the customer to the `Customer->new` method (e.g., `Customer->new(id => 'CUS-00123')`) and the `init` method would populate our object with the customer’s data. We could then either use this data in some way or alternatively alter it and use the `save` method to write the changed record back to the database.

2.4 Use UNIX “filter” model

UNIX filter programs give us a very good example to follow when it comes to building a number of small, reusable utilities each of which is designed to carry out one task.

2.4.1 Overview of the filter model

Many operating systems, principally UNIX and its variants, support a feature called I/O redirection. This feature is also supported in Microsoft Windows, although as it is a command line feature, it is not used as much as it is in UNIX. I/O redirection gives the user great flexibility over where a program gets its input and sends its output. This is achieved by treating all program input and output as file input and output. The operating system opens two special file handles called `STDIN` and `STDOUT`, which, by default, are attached to the user’s keyboard and monitor.⁵ This means that anything typed by the user on the keyboard appears to the program to be read from `STDIN` and anything that the program writes to `STDOUT` appears on the user’s monitor.

For example, if a user runs the UNIX command

```
ls
```

then a list of files in the current directory will be written to `STDOUT` and will appear on the user’s monitor.

There are, however a number of special character strings that can be used to redirect these special files. For example, if our user runs the command

```
ls > files.txt
```

then anything that would have been written to `STDOUT` is, instead, written to the file `files.txt`. Similarly, `STDIN` can be redirected using the `<` character. For example,

⁵ In practice there is also a third file handle called `STDERR` which is a special output file to which error messages are written, but this file can be safely ignored for the purposes of this discussion.

```
sort < files.txt
```

would sort our previously created file in lexical order (since we haven't redirected the output, it will go to the user's monitor).

Another, more powerful, concept is I/O pipes. This is where the output of one process is connected directly to the input of another. This is achieved using the `|` character. For example, if our user runs the command

```
ls | sort
```

then anything written to the `STDOUT` of the `ls` command (i.e., the list of files in the current directory) is written directly to the `STDIN` of the `sort` command. The `sort` command processes the data that appears on its `STDIN`, sorts that data, and writes the sorted data to its `STDOUT`. The `STDOUT` for the `sort` command has not been redirected and therefore the sorted list of files appears on the user's monitor.

A summary of the character strings used in basic I/O redirection is given in table 2.1. More complex features are available in some operating systems, but the characters listed are available in all versions of UNIX and Windows.

Table 2.1 Common I/O redirection

String	Usage	Description
<code>></code>	<code>cmd > file</code>	Runs <code>cmd</code> and writes the output to <code>file</code> , overwriting whatever was in <code>file</code> .
<code>>></code>	<code>cmd >> file</code>	Runs <code>cmd</code> and appends the output to the end of <code>file</code> .
<code><</code>	<code>cmd < file</code>	Runs <code>cmd</code> , taking input from <code>file</code> .
<code> </code>	<code>cmd1 cmd2</code>	Runs <code>cmd1</code> and passes any output as input to <code>cmd2</code>

2.4.2 Advantages of the filter model

The filter model is a very useful concept and is fundamental to the way that UNIX works. It means that UNIX can supply a large number of small, simple utilities, each of which do one task and do it well. Many complex tasks can be carried out by plugging a number of these utilities together. For example, if we needed to list all of the files in a directory with a name containing the string "proj01" and wanted them sorted in alphabetical order, we could use a combination of `ls`, `sort`, and `grep`⁶ like this:

```
ls -l | grep proj01 | sort
```

⁶ Which takes a text string as an argument and writes to `STDOUT` only input lines that contain that text.

Most UNIX utilities are written to support this mode of usage. They are known as *filters* as they read their input from `STDIN`, filter the data in a particular way, and write what is left to `STDOUT`.

This is a concept that we can make good use of in our data munging programs. If we write our programs so that they make no assumptions about the files that they are reading and writing (or, indeed, whether they are even reading from and writing to files) then we will have written a useful generic tool, which can be used in a number of different circumstances.

Example: I/O independence

Suppose, for example, that we had written a program called `data_munger` which munged data from one system into data suitable for use in another. Originally, we might take data from a file and write our output to another. It might be tempting to write a program that is called with two arguments which are the names of the input and output files. The program would then be called like this:

```
data_munger input.dat output.dat
```

Within the script we would open the files and read from the input, munge the data, and then write to the output file. In Perl, the program might look something like:

```
#!/usr/bin/perl -w

use strict;

my ($input, $output) = @ARGV;
open(IN, $input) || die "Can't open $input for reading: $!";
open(OUT, ">$output") || die "Can't open $output for writing: $!";

while (<IN>) {
    print OUT munge_data($_);
}
close(IN) || die "Can't close $input: $!";
close(OUT) || die "Can't close $output: $!";
```

This will certainly work well for as long as we receive our input data in a file and are expected to write our output data to another file. Perhaps at some point in the future, the programmers responsible for our data source will announce that they have written a new program called `data_writer`, which we should now use to extract data from their system. This program will write the extracted data to its `STDOUT`. At the same time the programmers responsible for our data sink announce a new program called `data_reader`, which we should use to load data into their system and which reads the data to be loaded from `STDIN`.

In order to use our program unchanged we will need to write some extra pieces of code in the script which drives our program. Our program will need to be called with code like this:

```
data_writer > input.dat
data_munger input.dat output.dat
data_reader < output.dat
```

This is already looking a little kludgy, but imagine if we had to make these changes across a large number of systems. Perhaps there is a better way to write the original program.

If we had assumed that the program reads from `STDIN` and writes to `STDOUT`, the program actually gets simpler and more flexible. The rewritten program looks like this:

```
#!/usr/bin/perl -w
while (<STDIN>) {
    print munge_data($_);
}
```

Note that we no longer have to open the input and output files explicitly, as Perl arranges for `STDIN` and `STDOUT` to be opened for us. Also, the default file handle to which the `print` function writes is `STDOUT`; therefore, we no longer need to pass a file handle to `print`. This script is therefore much simpler than our original one.

When we're dealing with input and output data files, our program is called like this:

```
data_munger < input.dat > output.dat
```

and once the other systems want us to use their `data_writer` and `data_reader` programs, we can call our program like this:

```
data_writer | data_munger | data_reader
```

and everything will work exactly the same without any changes to our program. As a bonus, if we have to cope with the introduction of `data_writer` before `data_reader` or vice versa, we can easily call our program like this:

```
data_writer | data_munger > output.dat
```

or this:

```
data_munger < input.dat | data_reader
```

and everything will still work as expected.

Rather than using the `STDIN` file handle, Perl allows you to make your program even more flexible with no more work, by reading input from the null file handle like this:

```
#!/usr/bin/perl -w
while (<>) {
    print munged_data($_);
}
```

In this case, Perl will give your program each line of every file that is listed on your command line. If there are no files on the command line, it reads from `STDIN`. This is exactly how most UNIX filter programs work. If we rewrote our `data_munger` program using this method we could call it in the following ways:

```
data_munger input.dat > output.dat
data_munger input.dat | data_reader
```

in addition to the methods listed previously.

Example: I/O chaining

Another advantage of the filter model is that it makes it easier to add new functionality into your processing chain without having to change existing code. Suppose that a system is sending you product data. You are loading this data into the database that drives your company’s web site. You receive the data in a file called `products.dat` and have written a script called `load_products`. This script reads the data from `STDIN`, performs various data munging processes, and finally loads the data into the database. The command that you run to load the file looks like this:

```
load_products < products.dat
```

What happens when the department that produces `products.dat` announces that because of a reorganization of their database they will be changing the format of your input file? For example, perhaps they will no longer identify each product with a unique integer, but with an alphanumeric code. Your first option would be to rewrite `load_products` to handle the new data format, but do you really want to destabilize a script that has worked very well for a long time? Using the UNIX filter model, you don’t have to. You can write a new script called `translate_products` which reads the new file format, translates the new product code to the product identifiers that you are expecting, and writes the records in the original format to `STDOUT`. Your existing `load_products` script can then read records in the format that it accepts from `STDIN` and can process them in exactly the same way that it always has. The command line would look like this:

```
translate_products < products.dat | load_products
```

This method of working is known as *chain extension* and can be very useful in a number of areas.

In general, the UNIX filter model is very powerful and often actually simplifies the program that you are writing, as well as making your programs more flexible. You should therefore consider using it as often as possible.

2.5 *Write audit trails*

When transforming data it is often useful to keep a detailed audit trail of what you have done. This is particularly true when the end users of the transformed data question the results of your transformation. It is very helpful to be able to trace through the audit log and work out exactly where each data item has come from. Generally, problems in the output data can have only one of two sources, either errors in the input data or errors in the transformation program. It will make your life much easier if you can quickly work out where the problem has arisen.

2.5.1 *What to write to an audit trail*

At different points in the life of a program, different levels of auditing will be appropriate. While the program is being developed and tested it is common practice to have a much more detailed audit trail than when it is being used day to day in a production environment. For this reason, it is often useful to write auditing code that allows you to generate different levels of output depending on the value of a variable that defines the audit level. This variable might be read from an environment variable like this:

```
my $audit_level = $ENV{AUDIT_LEVEL} || 0;
```

In this example we set the value of `$audit_level` from the environment variable `AUDIT_LEVEL`. If this level is not set then we default to 0, the minimum level. Later in the script we can write code like:

```
print LOG 'Starting processing at ', scalar localtime, "\n"
  if $audit_level > 0;
```

to print audit information to the previously opened file handle, `LOG`.

Standards for audit trails will typically vary from company to company, but some things that you might consider auditing are:

- start and end times of the process
- source and sink parameters (filenames, database connection parameters, etc.)
- ID of every record processed
- results of each data translation
- final count of records processed

2.5.2 Sample audit trail

A useful audit trail for a data munging process that takes data from a file and either creates or updates database records might look like this:

```
Process: daily_upd started at 00:30:00 25 Mar 2000
Data source: /data/input/daily.dat
Data sink: database customer on server DATA_SERVER (using id 'maint')
Input record: D:CUS-00123
Action: Delete
Translation: CUS-00123 = database id 2364
Record 2364 deleted successfully
Input record: U:CUS-00124:Jones & Co| [etc ...]
Action: Update
Translation: CUS-00124 = database id 2365
Record 2365 updated successfully
Input record: I:CUS-01000:Magnum Solutions Ltd| [etc ...]
Action: Insert
Integrity Check: CUS-01000 does not exist on database
Record 3159 inserted successfully

[many lines omitted]

End of file on data source
1037 records processed (60 ins, 964 upd, 13 del)
Process: daily_upd complete at 00:43:14 25 Mar 2000
```

2.5.3 Using the UNIX system logs

Sometimes you will want to log your audit trail to the UNIX system log. This is a centralized process in which the administrator of a UNIX system can control where the log information for various processes is written. To access the system log from Perl, use the `Sys::Syslog` module. This module contains four functions called `openlog`, `closelog`, `setlogmask`, and `syslog` which closely mirror the functionality of the UNIX functions with the same names. For more details on these functions, see the `Sys::Syslog` module's documentation and your UNIX manual. Here is an example of their use:

```
use Sys::Syslog;

openlog('data_munger.pl', 'cons', 'user');

# then later in the program
syslog('info', 'Process started');

# then later again
closelog();
```

Notice that as the system logger automatically timestamps all messages, we don't need to print the start time in our log message.

2.6 *Further information*

For more information on writing objects in Perl see *Object Oriented Perl* by Damian Conway (Manning).

For more information about the UNIX filter model and other UNIX programming tricks see *The UNIX Programming Environment* by Brian Kernigan and Rob Pike (Prentice Hall) or *UNIX Power Tools* by Jerry Peek, Tim O'Reilly, and Mike Loukides (O'Reilly).

For more general programming advice see *The Practice of Programming* by Brian Kernigan and Rob Pike (Addison-Wesley) and *Programming Pearls* by Jon Bentley (Addison-Wesley).

2.7 *Summary*

- Decoupling the various stages of your program can cut down on the code that you have to write by making code more reusable.
- Designing data structures carefully will make your programs more flexible.
- Write modules or objects to encapsulate your business rules.
- The UNIX filter model can make your programs I/O independent.
- Always write audit logs.