ajf@afco.demon.co.uk

Tutorial.pod

Table of Contents
Tutorial.pod

NAME

MARC::Doc::Tutorial - A documentation-only module for new users of MARC::Record

SYNOPSIS

```
 perldoc MARC::Doc::Tutorial
```

INTRODUCTION

What is MARC?

The MAchine Readable Cataloging format was designed by the Library of Congress in the late 1960s in order to allow libraries to convert their card catalogs into a digital format. The advantages of having computerized card catalogs were soon realized, and now MARC is being used by all sorts of libraries around the world to provide computerized access to their collections. MARC data in transmission format is optimized for processing by computers, so it‘s not very readable for the normal human. For more about the MARC format, visit the Library of Congress at http://www.loc.gov/marc/

What is this Tutorial?

The document you are reading is a beginners guide to using Perl to processing MARC data, written in the ‘cookbook’ style. Inside, you will find recipes on how to read, write, update and convert MARC data using the MARC::Record CPAN package. As with any cookbook, you should feel free to dip in at any section and use the recipe you find interesting. If you are new to Perl, you may want to read from the beginning.

The document you are reading is distributed with the MARC::Record package, however in case you are reading it somewhere else, you can find the latest version at CPAN:

http://www.cpan.org/modules/by-module/MARC/. You‘ll notice that some sections aren‘t filled in yet, which is a result of this document being a work in progress. If you have ideas for new sections please make a suggestion to perl4lib: http://www.rice.edu/perl4lib/.

History of MARC on CPAN

In 1999, a group of developers began working on MARC.pm to provide a Perl module for working with MARC data. MARC.pm was quite successful since it grew to include many new options that were requested by the Perl/library community. However, in adding these features the module swiftly outgrew its own clothes, and maintenance and addition of new features became extremely difficult. In addition, as libraries began using MARC.pm to process large MARC data files (1000 records) they noticed that memory consumption would skyrocket. Memory consumption became an issue for large batches of records because MARC.pm‘s object model was based on the ‘batch’ rather than the record... so each record in the file would often be read into memory. There were ways of getting around this, but they were not obvious. Some effort was made to reconcile the two approaches (batch and record), but with limited success.

In mid 2001, Andy Lester released MARC::Record and MARC::Field which provided a much simpler and maintainable package for processing MARC data with Perl. As its name suggests, MARC::Record treats an individual MARC record as the primary Perl object, rather than having the object represent a given set of records. Instead of forking the two projects, the developers agreed to encourage use of the MARC::Record framework, and to work on enhancing MARC::Record rather than extending MARC.pm further. Soon afterwards, MARC::Batch was added, which allows you to read in a large data file without having to worry about memory consumption.

Brief Overview of MARC Classes

The MARC::Record package is made up of several separate packages. This can be somewhat confusing to people new to Perl, or Object Oriented Programming. However this framework allows easy extension, and is built to support new input/output formats as their need arises. For a good introduction to using the object oriented features of Perl, see the perlboot documentation that came with your version of Perl.

Here are the packages that get installed with MARC::Record:

MARC::Batch
: A convenience class for accessing MARC data contained in an external file.

MARC::Field

An object for representing the indicators and subfields of a single MARC field.

MARC::Lint
: An extension to check the validity of MARC records.

MARC::Record
: This primary class represents a MARC record, being a container for multiple MARC::Field objects.

MARC::Doc::Tutorial
: This document!

MARC::File
: A superclass for representing files of MARC data.

MARC::File::MicroLIF
: A subclass of MARC::File for working with data encoded in the MicroLIF format.

MARC::File::USMARC
: A subclass of MARC::File for working with data encoded in the USMARC format.

Help Wanted!

It‘s already been mentioned but it‘s worth mentioning again: MARC::Doc::Tutorial is a work in progress, and you are encouraged to submit any suggestions for additional recipes via the perl4lib mailing list at http://www.rice.edu/perl4lib. Also, the development group is always looking for additional developers with good ideas; if you are interested you can sign up at SourceForge: http://sourceforge.net/projects/marcpm/.

READING

Reading a record from a file

Let‘s say you have a USMARC record in ‘file.dat’ and you‘d like to read in the record and print out its title.

```
## Example R1

## create a MARC::Batch object.
use MARC::Batch;
my $batch = MARC::Batch(’USMARC’, ’file.dat’);

## get a MARC record from the MARC::Batch object.
## the $record will be a MARC::Record object.
my $record = $batch->next();

## print the title contained in the record.
print $record->title(),"\n";
```

Using the distribution‘s ‘t/camel.usmarc‘, your result should be:

```
ActivePerl with ASP and ADO / Tobias Martinsson.
```

Iterating through a batch file

Now imagine that ‘file.dat’ actually contains multiple records and we want to print the title for each of them. Our program doesn‘t have to change very much at all: we just need to add a loop around our call to `next()`.

```
## Example R2

## create a MARC::Batch object.
use MARC::Batch;
my $batch = MARC::Batch->new(’USMARC’,’file.dat’);

while (my $record = $batch->next()) {

  ## print the title contained in the record.
  print $record->title(),"\n";

```

```
 12   }
```

The call to the `next()` method at line 7 returns the next record from the file. `next()` returns `undef` when there are no more records left in the file, which causes the `while` loop to end. This is a useful idiom for reading in all the records in a file. Your results with ‘camel.usmarc’ should be:

```
ActivePerl with ASP and ADO / Tobias Martinsson.
Programming the Perl DBI / Alligator Descartes and Tim Bunce.
.
.
.
Cross-platform Perl / Eric F. Johnson.
```

Checking for errors

It is a good idea to get in the habit of checking for errors. MARC/Perl has been designed to help you do this. Calls to `next()` when iterating through a batch file will return `undef` when there are no more records to return... **AND** when an error was encountered (see the next recipe to subvert this). You probably want to make sure that you didn‘t abruptly stop reading a batch file because of an error.

```
  1   ## Example R3
  2
  3   ## create a MARC::Batch object.
  4   use MARC::Batch;
  5   my $batch = MARC::Batch->new(’USMARC’,’file.dat’);
  6
  7   ## get a marc record from the MARC::Batch object.
  8   ## $record will be a MARC::Record object.
  9   while ( my $record = $batch->next() ) {
 10       print $record->title(),"\n";
 11   }
 12
 13   ## make sure there weren’t any problems.
 14   if ( my @warnings = $batch->warnings() ) {
 15       print "\nWarnings were detected!\n", @warnings;
 16   }
```

The call to `warnings()` at line 14 will retrieve any warning messages and store them in `@warnings`. This allows you to detect when `next()` has aborted prematurely (before the end of the file has been reached). When a warning is detected, an explanation is sent to `STDERR`. By introducing an error into ‘camel.usmarc‘, we‘ll receive the following output to `STDOUT`:

```
Warnings were detected!
Invalid indicators "a0" forced to blanks in record 1 for tag 245
```

Recovering from errors

You may want to keep reading a batch file even after an error has been encountered. If so, you will want to turn strict mode off using the `strict_off()` method. You can also prevent warnings from being printed to `STDERR` using the `warnings_off()` method. By default, strict is on as a safety precaution to prevent you from using corrupt MARC data. Once off, you can turn both strict and warnings back on again with the `strict_on()` and `warnings_on()` methods.

```
  1   ## Example R4
  2
  3   use MARC::Batch;
  4   my $batch = MARC::Batch->new(’USMARC’, ’file.dat’);
  5   $batch->strict_off();
  6
  7   while ( my $record = $batch->next() ) {
  8      print $record->title(),"\n";
  9   }
 10
 11   ## make sure there weren’t any problems.
 12   if ( my @warnings = $batch->warnings() ) {
      print "\nWarnings were detected!\n", @warnings;
}
```

Introducing a second error to the ‘camel.usmarc’ file gives the following:

```
ActivePerl with ASP and ADO / Tobias Martinsson.
Programming the Perl DBI / Alligator Descartes and Tim Bunce.
.
.
.
Cross-platform Perl / Eric F. Johnson.

Warnings were detected!
Invalid indicators "a0" forced to blanks in record 1 for tag 245
Invalid indicators "a0" forced to blanks in record 5 for tag 245
```

Looking at a field

Our previous examples use MARC::Record‘s `title()` method to easily access the 245 field, but you will probably want programs that access lots of other MARC fields. MARC::Record‘s `field()` method gives you complete access to the data found in any MARC field. The `field()` method returns a MARC::Field object which can be used to access the data, indicators, and even the individual subfields. Our next example shows how this is done.

```
 1   ## Example R5
 2
 3   ## open a file.
 4   use MARC::Batch;
 5   my $batch = MARC::Batch->new('USMARC','file.dat');
 6
 7   ## read a record.
 8   my $record = $batch->next();
 9
10   ## get the 100 field as a MARC::Field object.
11   my $field = $record->field('100');
12   print "The 100 field contains: ",$field->as_string(),"\n";
13   print "The 1st indicator is ",$field->indicator(1),"\n";
14   print "The 2nd indicator is ",$field->indicator(2),"\n";
15   print "Subfield d contains: ",$field->subfield('d'),"\n";
```

Which results in something like:

```
The 100 field contains: Martinsson, Tobias, 1976-
The 1st indicator is 1
The 2nd indicator is
Subfield d contains: 1976-
```

As before, use a `while` loop to iterate through all the records in a batch.

Looking at repeatable fields

So how do you retrieve data from repeatable fields? The `field()` method can help you with this as well. In our previous example‘s line 11, the `field()` method was used in a *scalar* context, since the result was being assigned to the variable `$field`. However in a *list* context, `field()` will return all the fields in the record of that particular type. For example:

```
## Example R6

use MARC::Batch;
my $file = MARC::Batch->new('USMARC','file.dat');
my $record = $batch->next();

## get all the 650 fields (list context).
my @fields = $record->field('650');

```

```
10   ## examine each 650 field and print it out.
11   foreach my $field (@fields) {
12     print $field->as_string(),"\n";
13   }
```

Which prints out the following for the first record of ‘t/camel.usmarc‘:

```
Active server pages.
ActiveX.
```

Looking at a set of related fields

`field()` also allows you to retrieve similar fields using ’.’ as a wildcard.

```
 1   ## Example R7
 2
 3   use MARC::Batch;
 4   my $batch = MARC::Batch->new(’USMARC’,’file.dat’);
 5   my $record = $batch->next();
 6
 7   # retrieve all title fields in one shot.
 8   foreach my $field ($record->field(’2..’)) {
 9     print $field->tag(),’ contains ’,$field->as_string(),"\n";
10   }
```

Notice the shorthand in line 8 which compacts lines 7-13 of our previous example. Instead of storing the fields in an array, the `field()` still returns a list in the `for` loop. Line 9 uses the `tag()` method which returns the tag number for a particular MARC field, which is useful when you aren‘t certain what tag you are currently dealing with. Sample output from this recipe:

```
245 contains ActivePerl with ASP and ADO / Tobias Martinsson.
260 contains New York : John Wiley & Sons, 2000.
```

You can also return all tags for a specific record by using ’...’ in `field` (though, see the next recipe).

Looking at all the fields in a record

The last example in this section illustrates how to retrieve *all* the fields in a record using the `fields()` method. This method is similar to passing ’...’ as a wildcard (see our previous recipe for alternative access).

```
 1   ## Example R8
 2
 3   use MARC::Batch;
 4   my $file = MARC::Batch->new(’USMARC’,’file.dat’);
 5   my $record = $batch->next();
 6
 7   ## get all of the fields using the fields() method.
 8   my @fields = $record->fields();
 9
10   ## print out the tag, the indicators and the field contents.
11   foreach my $field (@fields) {
12     print
13       $field->tag(), " ",
14       defined $field->indicator(1) ? $field->indicator(1) : "",
15       defined $field->indicator(2) ? $field->indicator(2) : "",
16       " ", $field->as_string, " \n";
17   }
```

The above code would print the following for the first record of ‘t/camel.usmarc‘:

```
001  fol05731351
003  IMchF
.
.
.
```

```
300    xxi, 289 p. : ill. ; 23 cm. + 1 computer  laser disc (4 3/4 in.)
500    "Wiley Computer Publishing."
650  0 Perl (Computer program language)
630 00 Active server pages.
630 00 ActiveX.
```

CREATING

The examples in the Section 1 covered how to read in existing USMARC data in a file. Section 2 will show you how to create a MARC record from scratch. The techniques in this section would allow you to write programs which create MARC records that could then be loaded into an online catalog, or sent to a third party.

Creating a record

To create a new MARC record, you‘ll need to first create a MARC::Record object, add a leader (though MARC::Record can create leaders automatically if you don‘t specifically define one), and then create and add MARC::Field objects to your MARC::Record object. For example:

```
## Example C1

## create a MARC::Record object.
use MARC::Record;
my $record = MARC::Record->new();

## add the leader to the record. optional.
$record->leader('00903pam   2200265 a 4500');

## create an author field.
my $author = MARC::Field->new(
  '100',1,'',
    a => 'Logan, Robert K.',
    d => '1939-'
  );
$record->append_fields($author);

## create a title field.
my $title = MARC::Field->new(
  '245','1','4',
    a => 'The alphabet effect /',
    c => 'Robert K. Logan.'
  );
$record->append_fields($title);
```

The key to creating records from scratch is to use `append_fields()`, which adds a field to the end of the record. Since each field gets added at the end, it‘s up to you to order the fields the way you want. `insert_fields_before()` and `insert_fields_after()` are similar methods that allow you to define where the field gets added. These methods are covered in more detail below.

WRITING

Sections 1 and 2 showed how to read and create USMARC data. Once you know how to read and create, it becomes important to know how to write the USMARC data to disk in order to save your work. In these examples, we will create a new record and save it to a file called ‘record.dat’.

Writing records to a file

```
## Example W1

## create a MARC::Record object.
use MARC::Record;
my $record = MARC::Record->new();

## add the leader to the record. optional.
```

```
 8   $record->leader(’00903pam   2200265 a 4500’);
 9
10   ## create an author field.
11   my $author = MARC::Field->new(
12     ’100’,1,’’,
13       a => ’Logan, Robert K.’,
14       d => ’1939-’
15     );
16
17   ## create a title field.
18   my $title = MARC::Field->new(
19     ’245’,’1’,’4’,
20       a => ’The alphabet effect /’,
21       c => ’Robert K. Logan.’
22     );
23
24   $record->append_fields($author, $title);
25
26   ## open a filehandle to write to ’record.dat’.
27   open(OUTPUT, ’> record.dat’) or die $!;
28   print OUTPUT $record->as_usmarc();
29   close(OUTPUT);
```

The `as_usmarc()` method call at line 28 returns a scalar value which is the raw USMARC data for `$record`. The raw data is then promptly printed to the `OUTPUT` file handle. If you want to output multiple records to a file, simply repeat the process at line 28 for the additional records. Also of note is the `append_fields` method: unlike recipe C1 which called the method once for each field added, this recipe demonstrates that `append_fields` can accept multiple arguments.
Note to the curious: the `as_usmarc()` method is actually an alias to the MARC::File::USMARC `encode()` method. Having separate `encode()` methods is a design feature of the MARC class hierarchy, since it allows extensions to be built that translate MARC::Record objects into different data formats.

Debugging with `as_formatted()`

Since raw USMARC data isn‘t very easy for humans to read, it is often useful to be able to see the contents of your MARC::Record object represented in a ‘pretty’ way for debugging purposes. If you have a MARC::Record object you‘d like to pretty-print, use the `as_formatted()` method.

```
## Example W2

## create a MARC::Record object.
use MARC::Record;
my $record = MARC::Record->new();

$record->leader(’00903pam  2200265 a 4500’);

$record->append_fields(
 MARC::Field->new(’100’,’1’,’’, a=>’Logan, Robert K.’, d=>’1939-’),
 MARC::Field->new(’245’,’1’,’4’, a=>’The alphabet effect /’, c=>’Robert K.
);

## pretty print the record.
print $record->as_formatted(), "\n";
```

This code will pretty print the contents of the newly created record:

```
LDR 00903pam  2200265 a 4500
100 1  _aLogan, Robert K.
       _d1939-
245 14 _aThe alphabet effect /
       _cRobert K. Logan.
```

Notice on lines 9-12 how you can add a list of new fields by creating MARC::Field objects within a call to `append_fields()`. This is yet another shorthand method to those shown in recipes C1 and W1. For more pretty-printing capabilities, try `marcdump()` in our next recipe.

Debugging with marcdump()

If you have written USMARC data to a file (as in recipe W2) and you would like to verify that the data is stored correctly you can use the `marcdump` command line utility that was installed with the MARC::Record package:

```
% marcdump record.dat
record.dat
LDR 00122pam  2200049 a 4500
100 1  _aLogan, Robert K.
       _d1939-
245 14 _aThe alphabet effect /
       _cRobert K. Logan.

 Recs  Errs Filename
----- ----- --------
    1     0 record.dat
```

As you can see, this command results in the record being pretty printed to your screen (`STDOUT`) similarly to the `as_formatted` method from recipe W2. It is useful for verifying your USMARC data after it has been stored on disk. More details about debugging are found later in VALIDATING.

UPDATING

Now that you know how to read, write and create MARC data, you have the tools you need to update or edit exiting MARC data. Updating MARC data is a common task for library catalogers. Sometimes there are huge amounts of records that need to be touched up... and while the touch ups are very detail oriented, they are also highly repetitive. Luckily, computers are tireless, and not very prone to error (assuming the programmer isn‘t).

When libraries receive large batches of MARC records for electronic text collections such as NetLibrary, Making of America, or microfiche sets like Early American Imprints, the records are often loaded into an online system and then the system is used to update the records. Unfortunately, not all these systems are created equal, and catalogers have to spend a great deal of time touching up each individual record. An alternative would be to process the records prior to import and then, once in the system, the records would not need editing. This scenario would save a great deal of time for the cataloger who would be liberated to spend their time doing original cataloging... which computers are notably bad at!

Adding a field

Imagine a batch of records in ‘file.dat’ that you‘d like to add local notes (590) to, then saving your changes:

```
## Example U1

## create our MARC::Batch object.
use MARC::Batch;
my $batch = MARC::Batch->new('USMARC','file.dat');

## open a file handle to write to.
open(OUT,'>new.dat') or die $!;

## read each record, modify, then print.
while ( my $record = $batch->next() ) {

    ## add a 590 field.
    $record->append_fields(
       MARC::Field->new('590','','',a=>'Access provided by Enron.')
    );

    print OUT $record->as_usmarc();

}

close(OUT);
```

Preserving field order

As its name suggests, `append_fields()` will add the 590 field in recipe U1 to the end of the record. If you want to preserve a particular order, you can use the `insert_fields_before()` and `insert_fields_after()` methods. In order to use these, you need to locate the field you want to insert before or after. Here is an example (`insert_fields_after()` works similarly):

```
## Example U2

use MARC::Batch;
my $batch = MARC::Batch->new('USMARC','file.dat');
open(OUT,'>new.dat') or die $!;

## read in each record.
while ( my $record = $batch->next() ) {

    ## find the tag after 590.
    my $before;
    foreach ($record->fields()) {
        $before = $_;
        last if $_->tag() > 590;
    }

    ## create the 590 field.
    my $new = MARC::Field->new('590','','',a=>'Access provided by Enron.');

    ## insert our 590 field after the $before.
    $record->insert_fields_before($before,$new);

    ## and print out the new record.
    print OUT $record->as_usmarc();

}
```

Deleting a field

You can also delete fields that you don‘t want. But you will probably want to check that the field contains what you expect before deleting it. Let‘s say Enron has gone out of business and the 590 field needs to be deleted:

```
## Example U3

use MARC::Batch;
my $batch = MARC::Batch->new('USMARC','new.dat');
open(OUT,'>newer.dat') or die $1;

while ( my $record = $batch->next() ) {

  ## get the 590 record.
  my $field = $record->field('590');

  ## if there is a 590 AND it has the word "Enron"...
  if ($field and $field->as_string() =~ /Enron/i) {

    ## delete it!
    $record->delete_field($field);

  }
19
20     ## output possibly modified record.
21     print OUT $record->as_usmarc();
22
23   }
```

The 590 field is retrieved on line 10, but notice how we check that we actually received a valid `$field`, and that it then contains the word ‘Enron’ before we delete it. You need to pass `delete_field()` a MARC::Field object that can be retrieved with the `field()` method.

Changing existing fields

Perhaps rather than adding or deleting a field, you need to modify an existing field. This is achieved in several steps: first, read in the MARC record you want to update, and then the field you‘re interested in. From there, call the field‘s `update` or `replace_with` methods to modify its contents, and then resave the record. Below is an example of updating existing 590 field‘s containing the word ‘Enron’ to indicate that access is now provided through Arthur Andersen:

```
 1   ## Example U4
 2
 3   use MARC::Batch;
 4   my $batch = MARC::Batch->new('USMARC','new.dat');
 5   open(OUT,'>newer.dat') or die $1;
 6
 7   while ( my $record = $batch->next() ) {
 8
 9     ## look for a 590 containing "Enron"...
10     my $field = $record->field('590');
11     if ($field and $field->as_string =~ /Enron/i) {
12
13       ## create a new 590 field.
14       my $new_field = MARC::Field->new(
15         '590','','', a => 'Access provided by Arthur Andersen.' );
16
17       ## replace existing with our new one.
18       $field->replace_with($new_field);
19
20     }
21
22     ## output possibly modified record.
23     print OUT $record->as_usmarc();
24
25   }
```

In this example, we used MARC::Field‘s method `replace_with()` to replace an existing field in the record with a new field that we created. To use `replace_with()`, you need to retrieve the field you want to replace from a MARC::Record object (line 10), create a new field to replace the existing one with (lines 13-15), and then call the existing field‘s `replace_with()` method passing the new field as an argument (lines 18). You must pass `replace_with()` a valid MARC::Field object.

Updating subfields and indicators

If you‘d rather not replace an existing field with a new one, you can also edit the contents of the field itself using the `update()` method. Let‘s say you‘ve got a batch of records and want to make sure that the 2nd indicator for the 245 field is properly set for titles that begin with ‘The’ (where the indicator should be ‘4’).

```
 1   ## Example U5
 2
 3   use MARC::Batch;
 4   my $batch = MARC::Batch->new('USMARC','file.dat');
 5   open(OUT,'>new.dat') or die $!;
 6
```

```
while (my $record = $batch->next()) {

  ## retrieve the 245 record.
  my $field_245 = $record->field('245');

  ## if we got 245 and it starts with 'The'...
  if ($field_245 and $field_245->as_string() =~ /^The /) {

    ## if the 2nd indicator isn't 4, update
    if ($field_245->indicator(2) != 4) {
      $field_245->update( ind2 => 4 );
    }

  }

  print OUT $record->as_usmarc();

}
```

In a similar fashion, you can update individual or multiple subfields:

```
  $field_245->update( a => 'History of the World :', b => 'part 1' );
```

But beware, you can only update the first occurrence of a subfield using `update()`. If you need to do more finer grained updates, you are advised to build a new field and replace the existing field with `replace_with()`.

Changing a record‘s leader

The above procedure works for fields, but editing the leader requires that you use the `leader()` method. When called with no arguments, `leader()` will return the current leader, and when you pass a scalar value as an argument, the leader will be set to this value. This example shows how you might want to update position 6 of a records leader to reflect a computer file.

```
## Example U6

use MARC::Batch;
my $batch = MARC::Batch->new('USMARC','file.dat');
open(OUT,'>new.dat') or die $!;
my $record = $batch->next();

## get the current leader.
my $leader = $record->leader();

## replace position 6 with 'm'
substr($leader,6,1) = 'm';

## update the leader
$record->leader($leader);

## save the record to a file
print OUT $record->as_usmarc();
```

Modifying fields without indicators

MARC::Record and MARC::Field are smart and know that you don‘t have field indicators with tags less than 010. Here‘s an example of updating/adding an 005 field to indicate a new transaction time. For a little pizzazz, we use Perl‘s `localtime()` to generate the data we need for this field.

```
## Example U7

use MARC::Batch;
my $batch = MARC::Batch->new('USMARC','file.dat');
open(OUT,'>new.dat') or die $!;
```

```

while (my $record = $batch->next() ) {

  ## see if there is a 005 field.
  my $field_005 = $record->field('005');

  ## delete it if we find one.
  $record->delete_field($field_005) if $field_005;

  ## figure out the contents of our new 005 field.
  my ($sec,$min,$hour,$mday,$mon,$year) = localtime();
  $year += 1900; $mon += 1; # catering to offsets.
  my $datetime = sprintf("%4d%02d%02d%02d%02d%02d.0",
                          $year,$mon,$mday,$hour,$min,$sec);

  ## create a new 005 field using our new datetime.
  $record->append_fields( MARC::Field->new('005',$datetime) );

  ## save record to a file.
  print OUT $record->as_usmarc();

}
```

Reordering subfields

You may find yourself in the situation where you would like to programmatically reorder, and possibly modify, subfields in a particular field. For example, imagine that you have a batch of records that have 856 fields which contain subfields z, u, and possibly 3... in any order! Now imagine that you‘d like to standardize the subfield z, and reorder them so that subfield 3 precedes subfield z, which precedes subfield u. This is tricky but can be done in the following manner: read in a record, extract the existing 856 field, build a new 856 field based on the existing one, replace the existing field with your newly created version.

```
## Example U8

use MARC::Batch;
my $batch = MARC::Batch->new('USMARC','856.dat');
open(OUT,'>856_new.dat') or die $!;

while (my $record = $batch->next()) {

  my $existing = $record->field('856');

  ## make sure 856 exists.
  if ($existing) {

    ## our ordered subfields.
    my @subfields = ();

    ## if we have a subfield 3, add it.
    if (defined($existing->subfield('3'))) {
      push(@subfields,'3',$existing->subfield('3'));
    }

    ## now add subfields z and u.
    push(@subfields,'z','Access restricted',
      'u',$existing->subfield('u'));

    ## create a new 856.
    my $new = MARC::Field->new(
       856', $existing->indicator(1),
     $existing->indicator(2), @subfields
   );

   ## replace the existing subfield.
   $existing->replace_with($new);

  }

  ## write out the record
  print OUT $record->as_usmarc();

 }
```

Updating subject subfield x to subfield v

As a somewhat more complicated example, you may find yourself wanting to update the last subfield x in a 650 field to be a subfield v instead. With the MARC::Field `subfields()` and `replace_with()` methods along with some fancy footwork this can be done relatively easily.

```
## Example U9

use MARC::Batch;

my $file = shift;

my $batch = MARC::Batch->new('USMARC', $file);
while ( my $record = $batch->next() ) {

 # go through all 6XX fields in the record.
 foreach my $subject ( $record->field( '6..' ) ) {

  # extract subfields as an array of array refs.
  my @subfields = $subject->subfields();

  # setup an array to store our new field.
  my @newSubfields = ();

  # a flag to indicate that we found an subfield x.
  my $foundX = 0;

  # use pop() to read the subfields backwards.
  while ( my $subfield = pop( @subfields ) ) {

   # for convenience, pull out the subfield
   # code and data from  the array ref.
   my ($code,$data) = @$subfield;

   # if the subfield code is 'x' and
   # we haven't already found one...
   if ( $code eq 'x' and ! $foundX ) {

    # change to a v.
    $code = 'v';

    # set flag so we know not to
    # translate any more subfield x.
    $foundX = 1;

   }

   # add our (potentially changed) subfield
```

```
      # data to our new subfield data array.
      unshift( @newSubfields, $code, $data );

    }

    # if we did find a subfield x, then create a new field using our
    # new subfield data, and replace the old one with the new one.
    if ( $foundX ) {
      my $newSubject = MARC::Field->new(
        $subject->tag(),
        $subject->indicator(1),
        $subject->indicator(2),
        @newSubfields
      );
      $subject->replace_with( $newSubject );
    }

  }

  # output the potentially changed record as MARC.
  print $record->as_usmarc();

}
```

VALIDATING

The MARC::Record package has some extra goodies to allow you to validate records: MARC::Lint. MARC::Lint provides an extensive battery of tests, and it also provides a framework for adding more.

Using MARC::Lint

Here is an example of using MARC::Lint to generate a list of errors present in a batch of records in a file named ‘file.dat‘:

```
## Example V1

use MARC::Batch;
use MARC::Lint;

my $batch = MARC::Batch->new(’USMARC’,’file.dat’);
my $linter = MARC::Lint->new();
my $counter = 0;

while (my $record = $batch->next() ) {

  $counter++;

  ## feed the record to our linter object.
  $linter->check_record($record);

  ## get the warnings...
  my @warnings = $linter->warnings();

  ## output any warnings.
  if (@warnings) {

    print "RECORD $counter\n";
    print join("\n",@warnings),"\n";

  }

}
```

MARC::Lint is quite thorough, and will check the following when validating: presence of a 245 field, repeatability of fields and subfields, valid use of subfield within particular fields, presence of indicators and their values.

Customizing MARC::Lint

MARC::Lint makes no claim to check **everything** that might be wrong with a MARC record. In practice, individual libraries may have their own idea about what is valid or invalid. For example, a library may mandate that all MARC records with an 856 field should have a subfield z that reads "Connect to this resource".

MARC::Lint does provide a framework for adding rules. It can be done using the object oriented programming technique of inheritance. In short, you can create your own subclass of MARC::Lint, and then use it to validate your records. Here‘s an example:

```
## Example V2

## first, create our own subclass of MARC::Lint.
## should be saved in a file called MyLint.pm.

package MyLint;
use base qw(MARC::Lint);

## add a method to check that the 856
## fields contain a correct subfield z.
sub check_856 {

  ## your method is passed the MARC::Lint
  ## and MARC::Field objects for the record.
  my ($self,$field) = @_;

  if ($field->subfield(’z’) ne ’Connect to this resource’) {

    ## add a warning to our lint object.
    $self->warn("856 subfield z must read ’Connect to this resource’.");

  }

}
```

Then create a separate program that uses your subclass to validate your MARC records. You‘ll need to make sure your program is able to find your module (in this case, MyLint.pm)... this can be achieved by putting both MyLint.pm and the following program in the same directory:

```
## Example V3

use MARC::Batch;
use MyLint;

my $linter = MyLint->new();
my $batch = MARC::Batch->new(’USMARC’,’file.marc’);
my $counter = 0;

while (my $record = $batch->next()) {

  $counter++;

  ## check the record
  $linter->check_record($record);

  ## get the warnings, and print them out
  my @warnings = $linter->warnings();
  if (@warnings) {
    print "RECORD $counter\n";
```

```
 21        print join("\n",@warnings),"\n";
 22      }
 23
 24    }
```

Notice how the call to `check_record()` at line 15 automatically calls the `check_record` in MARC::Lint. The property of inheritance is what makes this happen. `$linter` is an instance of the MyLint class, and MyLint inherits from the MARC::Lint class, which allows `$linter` to inherit all the functionality of a normal MARC::Lint object **plus** the new functionality found in the `check_856` method.

Notice also that we don‘t have to call `check_856()` directly. The call to `check_record()` automatically looks for any `check_XXX` methods that it can call to verify the record. Pretty neat stuff.

If you‘ve added validation checks that you think could be of use to the general public, please share them on the perl4lib mailing list, or become a developer and add them to the source!

SWOLLEN APPENDICES

Brian Eno fans might catch this reference to his autobiography which was comprised of a years worth of diary entries plus extra topics at the end, and was entitled "A Year With Swollen Appendices". The following section is a grab bag group of appendices. Many of them are not filled in yet; this is because they are just ideas... so perhaps the appendices aren‘t that swollen yet. Feel free to suggest new ones, or to fill these in.

Comparing Collections

Authority Records

URLs

ISBN/ISSNs

Call numbers

Subject headings

Suppose you have a batch of MARC records and you want to extract all the subject headings, generating a report of how many times each subject heading appeared in the batch:

```
use MARC::File::USMARC;
use constant MAX => 20;

my %counts;

my $filename = shift or die "Must specify filename\n";
my $file = MARC::File::USMARC->in( $filename );

while ( my $marc = $file->next() ) {
    for my $field ( $marc->field("6..") ) {
        my $heading = $field->subfield('a');

        # trailing whitespace / punctuation.
        $heading =~ s/[.,]?\s*$//;

        # Now count it.
        ++$counts{$heading};
    }
}
$file->close();

# Sort the list of headings based on the count of each.
my @headings = reverse sort { $counts{$a} <=> $counts{$b} } keys %counts;

# Take the top N hits...
@headings = @headings[0..MAX-1];

```

```
# And print out the results.
for my $heading ( @headings ) {
    printf( "%5d %s\n", $counts{$heading}, $heading );
}
```

Which will generate results like this:

```
600 United States
140 World War, 1939-1945
 78 Great Britain
 63 Afro-Americans
 61 Indians of North America
 58 American poetry
 55 France
 53 West (U.S.)
 53 Science fiction
 53 American literature
 50 Shakespeare, William
 48 Soviet Union
 46 Mystery and detective stories
 45 Presidents
 43 China
 40 Frontier and pioneer life
 38 English poetry
 37 Authors, American
 37 English language
 35 Japan
```

HTML

XML

Excel

Databases

Z39.50

Chris Biemesderfer was kind enough to contribute a short example of how to use MARC::Record in tandem with Net::Z3950. Net::Z3950 is a CPAN module which provides an easy to use interface to the Z39.50 protocol so that you can write programs that retrieve records from bibliographic database around the world.

Chris' program is a command line utility which you run like so:

```
./zm.pl 0596000278
```

where 0596000278 is an ISBN (for the 3rd edition of the Camel incidentally). The program will query the Library of Congress Z39.50 server for the ISBN, and dump out the retrieved MARC record on the screen. The program is designed to lookup multiple ISBNs if you separate them with a space. This is just an example showing what is possible.

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

# GET-MARC-ISBN -- Get MARC records by ISBN from a Z39.50 server

use strict;
use Carp;
use Net::Z3950;
use MARC::Record;

exit if ($#ARGV < 0);

# We handle multiple ISBNs in the same query by assembling a
# (potentially very large) search string with Prefix Query Notation
# that ORs the ISBN-bearing attributes.
```

```
#
# For purposes of automation, we want to request batches of many MARC
# records.  I am not a Z39.50 weenie, though, and I don’t know
# offhand if there is a limit on how big a PQN query can be...

my $zq = "\@attr 1=7 ". pop();
while (@ARGV) { $zq = ’@or @attr 1=7 ’. pop() ." $zq" }

## HERE IS THE CODE FOR Z3950 REC RETRIEVAL
# Set up connection management structures, connect
# to the server, and submit the Z39.50 query.

my $mgr = Net::Z3950::Manager->new( databaseName => ’voyager’ );
$mgr->option( elementSetName => "f" );
$mgr->option( preferredRecordSyntax => Net::Z3950::RecordSyntax::USMARC );

my $conn = $mgr->connect(’z3950.loc.gov’, ’7090’);
croak "Unable to connect to server" if !defined($conn);

my $rs = $conn->search($zq);

my $numrec = $rs->size();
print STDERR "$numrec record(s) found\n";

for (my $ii = 1; $ii <= $numrec; $ii++) {

    # Extract MARC records from Z3950
    # result set, and load MARC::Record.
    my $zrec = $rs->record($ii);
    my $mrec = MARC::Record->new_from_usmarc($zrec->rawdata());
    print $mrec->as_formatted, "\n\n";

}
```

Procite/Endnote

CONTRIBUTORS

Many thanks to all the contributors who have made this document possible.

- Chris Biemesderfer <chris@seagoat.com>
- Morbus Iff <morbus@disobey.com>
- Andy Lester <andy@petdance.com>
- Christopher Morgan <morgan@acm.org>
- Shashi Pinheiro <SPinheiro@utsa.edu>
- Jackie Shieh <jshieh@umich.edu>
- Ed Summers <ehs@pobox.com>