POD Translation by *pod2pdf*

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 let the make a suggestion on 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 it‘s own clothes, and maintenance and addition of new features became extremely difficult.

In mid 2001 Andy Lester released MARC::Record and MARC::Field which provided a much more simpler and maintainable package for processing MARC data with Perl. 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.

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:

```
perldoc perlboot
```

Here are the packages that get installed when you install the MARC::Record patch.

MARC::Record
: The primary class, which represents the data held in a MARC record. It is basically a container class for multiple MARC::Field objects.

MARC::Field
: An object for representing the indicators and subfields contained in a single MARC field.

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

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

MARC::File::USMARC

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

MARC::File::MicroLIF

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

MARC::Doc::Tutorial

This document!

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 listserv 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 a file called 'file.dat' and you would like to read in the record and print out it's title.

```
 1   ## Example 1
 2
 3   ## create a MARC::Batch object
 4   use MARC::Batch;
 5   my $batch = MARC::Batch( 'USMARC', 'file.dat');
 6
 7   ## get a marc record from the MARC::Batch object
 8   ## $record will be a MARC::Record object
 9   my $record = $batch->next();
10
11   ## print the title contained in the record
12   print $record->title(),"\n";
```

Iterating through a batch file

Now imagine that 'file.dat' actually contains multiple records and we want to print out the title for all 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()`.

```
 1   ## Example 2
 2
 3   ## create a MARC::Batch object
 4   use MARC::Batch;
 5   my $batch = MARC::Batch->new('USMARC','file.dat');
 6
 7   while (my $record = $file->next()) {
 8
 9     ## print the title contained in the record
10     print $record->title(),"\n";
11
12   }
13
14   ## we're done so close the file
15   $file->close();
```

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.

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. You probably want to make sure that you didn't

abruptly stop reading a batch file because of an error.

```
 1   ## Example 3
 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();
11   }
12
13   ## make sure there weren't any problems
14   my @warnings = $batch->warnings();
15   if ( @warnings ) {
16       print "we found some warnings!", join("\n",@warnings);
17   }
```

The call to warnings() at line 14 will retrieve any warning messages and store them in the array @warnings. This allows you to detect when next() has aborted prematurely (before the end of the file has been reached).

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.

```
 1   ## Example 4
 2
 3   use MARC::Batch;
 4   my $batch = MARC::Batch->new( 'USMARC', 'file.dat' );
 5   $batch->strict_off();
 6   $batch->warnings_off();
 7
 8   while ( my $record = $batch->next() ) {
 9      print $record->title();
10   }
```

Use of strict_off() allows you to continue reading after an error is encountered. 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.

Looking at a field

Examples 1.1 - 1.3 use MARC::Record's title() method to easily access the 245 field...but you probably will want to write programs that access lots of other MARC fields.

MARC::Record's field() method gives you complete access 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. Example 1.4 shows you how this is done.

```
 1   ## Example 4
 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 record's 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";
```

Looking at repeatable fields

So how do you retrieve data from repeatable fields? The field() method can help you with this as well. Above in example 1.4 on 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:

```
1   ## Example 5
2
3   use MARC::Batch;
4   my $file = MARC::Batch->new('USMARC','file.dat');
5   my $record = $batch->next();
6
7   ## get all the 650 fields (list context)
8   my @fields = $record->field('650');
9
10  ## examine each 650 field and print it out
11  foreach my $field (@fields) {
12    print $field->as_string(),"\n";
13  }
```

Looking at a set of related fields

field() also allows you to retrieve similar fields using '.' as a wildcard. For example, this functionality allows you to retrieve all the note fields in one shot.

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

Notice the shorthand in line 7 which compacts lines 7-13 of Example 1.5. Instead of storing the fields in an array, the field() still returns a list in the for loop. Line 8 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 dealing with.

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.

```
1   ## Example 7
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 useing 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       $field->indicator(1),
15       $field->indicator(2), " ",
16       $field->as_string, " ",
17       "\n";
18   }
```

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 that create MARC records that could then be loaded into an online catalog, or sent to a third party.

Creating a record

To create a record you need to:

1 Create a MARC::Record object.

2 Add a leader to the record.

3 Create MARC::Field objects for each field you want to have in the record.

4 Add each of the MARC::Field objects to the MARC::Record object.

For example:

```
 1   ## Example 8
 2
 3   ## create a MARC::Record object
 4   use MARC::Record;
 5   my $record = MARC::Record->new();
 6
 7   ## add the leader to the record
 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   $record->append_fields($author);
17
18   ## create a title field
19   my $title = MARC::Field->new(
20     '245','1','4',
21       a => 'The alphabet effect /',
22       c => 'Robert K. Logan.'
23     );
24   $record->append_fields($title);
```

The key to creating records from scratch is to use the append_fields() method, 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.

Creating a record from raw MARC data in a variable

The above examples illustrated how to create a record from MARC data stored on disk. However you may have the raw USMARC data stored in a variable and want to create a MARC::Record from it. This

situation can arise when you are able to pull the MARC data out of a database, or using some input method other that the filesystem. If you ever find yourself in this position take a look at MARC::Record‘s new_from_usmarc() method which allows you to create a MARC::Record object from the USMARC data stored in a variable.

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 this example we will create a new record and save it to a file called ‘record.dat’.

Writing records to a file

```
 1   ## Example 9
 2
 3   ## create MARC object
 4   use MARC::Record;
 5   my $record = MARC::Record->new();
 6   $record->leader('00903pam  2200265 a 4500');
 7   my $author = MARC::Field->new('100','1','',
 8      a=>'Logan, Robert K.', d=>'1939-'
 9      );
10   my $title = MARC::Field->new('245','1','4',
11      a=>'The alphabet effect /', c=>'Robert K. Logan.'
12      );
13   $record->append_fields($author,$title);
14
15   ## open a filehandle to write to 'file.dat'
16   open(OUTPUT, '> record.dat');
17   print OUTPUT $record->as_usmarc();
18   close(OUTPUT);
```

The as_usmarc() method call at line 17 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 you could simply repeat the process at line 17 for the additional records.
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 MARC::Record object you‘d like to pretty-print use the as_formatted() method.

```
 1   ## Example 10
 2
 3   ## create MARC object
 4   use MARC::Record;
 5   my $record = MARC::Record->new();
 6   $record->leader('00903pam  2200265 a 4500');
 7   $record->append_fields(
 8      MARC::Field->new('100','1','',
 9         a=>'Logan, Robert K.', d=>'1939-'
10         ),
11      MARC::Field->new('245','1','4',
12         a=>'The alphabet effect /', c=>'Robert K. Logan.'
13         ),
14      );
15
16   ## pretty print the record
```

```
17   print $record->as_formatted();
```

Unlike example 9 this code will pretty print the contents of the newly created record to the screen. Notice on lines to how you can add a list of new fields by creating MARC::Field objects within a call to append_fields().

Debugging with marcdump()

If you have written USMARC data to a file (as in example 9) and you would like to verify that the data is stored correctly you can use the marcdump command line utility that was installed when you installed 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). 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 such as 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 touching up. 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 that you have a batch of records in a file called ‘file.dat’ and that you would like to add a local note to (590) to each record and save it as ‘file_2.dat’.

```
 1   ## Example 11
 2
 3   ## create our MARC::Batch object
 4   use MARC::Batch;
 5   my $batch = MARC::Batch->new('USMARC','file.dat');
 6
 7   ## open a file handle to write to
 8   open(OUT,'>new.dat');
 9
10   ## read in each record
11   while ( my $record = $batch->next() ) {
12
```

```
13       ## add a 590 field
14       $record->append_fields(
15          MARC::Field->new('590','','',a=>'Access provided by Enron.')
16       );
17
18     print OUT $record->as_usmarc();
19
20   }
21
22   close(OUT);
```

Notice on lines 3-5 how MARC::Batch is used instead of MARC::File::USMARC. MARC::Batch provides an alternate way of reading records from files, and provides a uniform interface to the different MARC::File modules.

Preserving field order

As its name suggests append_fields() will add the 590 field 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(). In order to use these you need to locate the field you want to insert before or after. Here is an example:

```
1   ## Example 12
2
3   use MARC::Batch;
4   my $batch = MARC::Batch->new('USMARC','file.dat');
5   open(OUT,'>new.dat');
6
7   ## read in each record
8   while ( my $record = $batch->next() ) {
9
10      ## find the first tag after 590
11      my $before;
12      foreach ($record->fields()) {
13          $before = $_;
14          last if $_->tag() > 590;
15      }
16
17      ## create the 590 field
18      my $new =
19          MARC::Field->new('590','','',a=>'Access provided by Enron.');
20
21      ## insert our 590 field
22      $record->insert_fields_before($before,$new);
23
24      print OUT $record->as_usmarc();
25
26  }
```

insert_fields_after() works in a simliar fashion to insert_fields_before() but with the expected change of behavior.

Deleting a field

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

```
1   ## Example 13
2
3   use MARC::Batch;
4   my $batch = MARC::Batch->new('USMARC','new.dat');
```

```
 5   open(OUT,'>newer.dat');
 6
 7   while ( my $record = $batch->next() ) {
 8
 9     ## get the 590 record
10     my $field = $record->field('590');
11
12     ## if there is a 590 field AND it has the word Enron in it
13     if ($field and $field->as_string() =~ /Enron/i) {
14
15       ## delete it!
16       $record->delete_field($field);
17
18     }
19
20     ## output possibly modified record to our new file
21     print OUT $record->as_usmarc();
22
23   }
```

The 590 field is retrieved on line 8; but notice how we check that we actually got a 590 field in $field, and that it 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:

1. Read in the MARC record that you want to update.
2. Retrieve the field you want to update.
3. Call the field's update() method or replace_with() method to modify the contents of the field.
4. Save the record.

Below is an example of updating any existing 590 field's containing the word 'enron' to indicate that access is now provided through Arthur Andersen.

```
 1   ## Example 14
 2
 3   use MARC::Batch;
 4   my $batch = MARC::Batch->new('USMARC','new.dat');
 5   open(OUT,'>newer.dat');
 6
 7   while ( my $record = $batch->next() ) {
 8
 9     ## look for and a 590 field 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','','',
16           a => 'Access provided by Arthur Andersen.'
17         );
18
19       ## replace existing 590 field with the our new one
20       $field->replace_with($new_field);
21
22     }
```

```
23
24      ## print out our (possibly) modified record
25      print OUT $record->as_usmarc();
26
27    }
```

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 7), create a new field to replace the existing one with (lines 13-17), and then call the existing field's replace_with() method passing the new field as an argument (lines 19-20). You must pass replace_with() a valid MARC::Field object for things to work.

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 you want to make sure that the 2nd indicator for the 245 field is properly set for titles that begin with 'The'. The 2nd indicator should be '4' for titles beginning with 'The'.

```
 1    ## Example 15
 2
 3    use MARC::Batch;
 4    my $batch = MARC::Batch->new('USMARC','file.dat');
 5    open(OUT,'>new.dat');
 6
 7    while (my $record = $batch->next()) {
 8
 9      ## retrieve the 245 record
10      my $field_245 = $record->field('245');
11
12      ## if we got the 245 and it starts with 'The'
13      if ($field_245 and $field_245->as_string() =~ /^The /) {
14
15        ## if the 2nd indicator isn't 4 we need to update
16        if ($field_245->indicator(2) != 4) {
17         $field_245->update( ind2 => 4 );
18        }
19
20      }
21
22      print OUT $record->as_usmarc();
23
24    }
```

The call to update() at line 17 sets the second indicator of the existing 245 field to 4. In a simliar fashion you can also 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

This 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 that the record is for a computer file.

```
## Example 16

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

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

 ## replace what is in 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 17

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

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

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

  ## delete it if we found it
  $record->delete_field($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;
  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
  $record->append_fields('005',$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 programatically 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 subfield 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:

1

Read in a record

2 Extract the existing 856 field.

3 Build a new 856 field based on the existing field.

4 Replace the existing 856 field with the new one.

5 Save our modified record.

Here is the example in detail:

```
1   ## Example 18
2
3   use MARC::Batch;
4   my $batch = MARC::Batch->new('USMARC','856.dat');
5   open(OUT,'>856_new.dat');
6
7   while (my $record = $batch->next()) {
8
9     my $existing = $record->field('856');
10
11    ## make sure the record has an 856 field we can edit
12    if ($existing) {
13
14      ## now we're going to build a list of our new subfields (in order)
15      my @subfields = ();
16
17      ## if the 856 field has a subfield 3 add it
18      if (defined($existing->subfield('3'))) {
19        push(@subfields,'3',$existing->subfield('3'));
20      }
21
22      ## now add subfields z and u
23      push(@subfields,'z','Access restricted',
24        'u',$existing->subfield('u'));
25
26      ## create a new 856 field using the new reordered subfields
27      my $new = MARC::Field->new(
28        '856', $existing->indicator(1), $existing->indicator(2), @subfields
29      );
30
31      ## replace the existing subfield with our new one
32      $existing->replace_with($new);
33
34    }
35
36    ## write out the record
37    print OUT $record->as_usmarc();
38
39  }
```

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'.

```
1   ## Example 19
2
3   use MARC::Batch;
4   use MARC::Lint;
5
6   my $batch = MARC::Batch->new('USMARC','file.dat');
```

```
 7   my $linter = MARC::Lint->new();
 8   my $counter = 0;
 9
10   while (my $record = $batch->next() ) {
11
12     $counter++;
13
14     ## feed the record to our linter object
15     $linter->check_record($record);
16
17     ## get the warnings
18     my @warnings = $linter->warnings();
19
20     ## output warnings (if any) with the record #
21     if (@warnings) {
22
23       print "RECORD $counter\n";
24       print join("\n",@warnings),"\n";
25
26     }
27
28   }
```

MARC::Lint is quite thorough, and will check the following when validating:

- Presence of 245 field.
- Repeatability of fields.
- Repeatability of subfields.
- Valid use of subfield within particular fields.
- Presence of indicators.
- Indicator 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:

```
 1   ## Example 20
 2
 3   ## first, create our own subclass of MARC::Lint
 4   ## should be saved in a file called MyLint.pm
 5
 6   package MyLint;
 7   use base qw(MARC::Lint);
 8
 9   ## add a method to check that the 856 fields contain
10   ## a correct subfield z
11   sub check_856 {
12
13     ## your method is passed the MARC::Lint and MARC::Field objects
14     my ($self,$field) = @_;
15
16     if ($field->subfield(’z’) ne ’Connect to this resource’) {
17
```

```
18        ## add a warning to our lint object
19        $self->warn("856 subfield z must read 'Connect to this resource'.");
20
21      }
22
23    }
24
25
```

```
 1    ## Then create a separate program that uses your subclass to validate
 2    ## NOTE: you need to make sure your program is able to find your
 3    ## module MyLint.pm ... this can be achieved by putting both MyLint.pm
 4    ## and this program in the same directory
 5
 6    use MARC::Batch;
 7    use MyLint;
 8
 9    my $linter = MyLint->new();
10    my $batch = MARC::Batch->new('USMARC','file.marc');
11    my $counter = 0;
12
13    while (my $record = $batch->next()) {
14
15      $counter++;
16
17      ## check the record
18      $linter->check_record($record);
19
20      ## get the warnings, and print them out
21      my @warnings = $linter->warnings();
22      if (@warnings) {
23        print "RECORD $counter\n";
24        print join("\n",@warnings),"\n";
25      }
26
27    }
```

Notice how the call to check_record() at line 18 just above 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() directy. 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 general public please share them on the perl4lib listserv, 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 probably 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, and generate a report of how many times each subject heading appeared in the batch.

```
 1   use MARC::File::USMARC;
 2   use constant MAX => 20;
 3
 4   my %counts;
 5
 6   my $filename = shift or die "Must specify filename\n";
 7   my $file = MARC::File::USMARC->in( $filename );
 8
 9   while ( my $marc = $file->next() ) {
10       for my $field ( $marc->field("6..") ) {
11           my $heading = $field->subfield('a');
12
13           # Remove certain trailing whitespace and punctuation.
14           $heading =~ s/[.,]?\s*$//;
15
16           # Now count it
17           ++$counts{$heading};
18       }
19   }
20   $file->close();
21
22   # Sort the list of headings based on the count of each.
23   my @headings = reverse sort { $counts{$a} <=> $counts{$b} } keys %counts;
24
25   # Take the top N hits.
26   @headings = @headings[0..MAX-1];
27
28   # Print out the results
29   for my $heading ( @headings ) {
30       printf( "%5d %s\n", $counts{$heading}, $heading );
31   }
```

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 can 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 mutliple ISBNs if you separate them with a space. This is just an example showing what is possible.

```
 1  #! /usr/bin/perl -w
 2
 3  # GET-MARC-ISBN -- Get MARC records by ISBN from a Z39.50 server
 4
 5  use strict;
 6  use Carp;
 7  use Net::Z3950;
 8  use MARC::Record;
 9
10  exit if ($#ARGV < 0);
11
12  # We handle multiple ISBNs in the same query by assembling a
13  # (potentially very large) search string with Prefix Query Notation
14  # that ORs the ISBN-bearing attributes.
15  #
16  # For purposes of automation, we want to request batches of many MARC
17  # records.  I am not a Z39.50 weenie, though, and I don't know
18  # offhand if there is a limit on how big a PQN query can be...
19
20  my $zq = "\@attr 1=7 ". pop();
21  while (@ARGV) { $zq = '@or @attr 1=7 '. pop() ." $zq" }
22
23  ## HERE IS THE CODE FOR Z3950 REC RETRIEVAL
24
25  # Set up connection management structures, connect to the server,
26  # and submit the Z39.50 query.
27
28  my $mgr = Net::Z3950::Manager->new( databaseName => 'voyager' );
29  $mgr->option( elementSetName => "f" );
30  $mgr->option( preferredRecordSyntax => Net::Z3950::RecordSyntax::USMARC );
31
32  my $conn = $mgr->connect('z3950.loc.gov', '7090');
33  croak "Unable to connect to server $server" if !defined($conn);
34
35  my $rs = $conn->search($zq);
36
37  my $numrec = $rs->size();
38  print STDERR "$numrec record(s) found\n";
39
```

```
40   for (my $ii = 1; $ii <= $numrec; $ii++) {
41
42       # Extract MARC records from Z3950 result set, and load MARC::Record.
43
44       my $zrec = $rs->record($ii);
45       my $mrec = MARC::Record->new_from_usmarc($zrec->rawdata());
46       print $mrec->as_formatted, "\n\n";
47
48   }
```

Procite/Endnote

CONTRIBUTORS

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

- Chris Biemesderfer <chris@seagoat.com>
- Andy Lester <andy@petdance.com>
- Christopher Morgan <morgan@acm.org>
- Jackie Shieh <jshieh@umich.edu>
- Ed Summers <ehs@pobox.com>