MARC.pm

Table of Contents
MARC.pm

NAME

MARC.pm - Perl extension to manipulate MAchine Readable Cataloging records.

SYNOPSIS

```
use MARC;

      # constructors
$x=MARC->new();
$x=MARC->new("filename","fileformat");
$x->openmarc({file=>"makrbrkr.mrc",'format'=>"marcmaker",
              increment=>"5", lineterm=>"\n",
              charset=>\%char_hash});
$record_num=$x->createrecord({leader=>"00000nmm  2200000 a 4500"});

      # input/output operations
$y=$x->nextmarc(10);                  # increment
$x->closemarc();
print $x->marc_count();
$x->deletemarc({record=>'2',field=>'110'});
$y=$x->selectmarc(['4','21-50','60']);

      # character translation
my %inc = %{$x->usmarc_default()};    # MARCMaker input charset
my %outc = %{$x->ustext_default()};   # MARCBreaker output charset

      # data queries
@records = $x->searchmarc({field=>"245"});
@records = $x->searchmarc({field=>"260",subfield=>"c",
                           regex=>"/19../"});
@records = $x->searchmarc({field=>"245",notregex=>"/huckleberry/i"});
@results = $x->getvalue({record=>'12',field=>'856',subfield=>'u'});

      # header and control field operations
$rldr = $x->unpack_ldr($record);
print "Desc is $rldr->{Desc}";
next if ($x->bib_format($record) eq 'SERIALS');
$rff = $x->unpack_008($record);
last if ($rff->{'Date1'}=~/00/ or $rff->{'Date2'}=~/00/);

      # data modifications
$x->addfield({record=>"2", field=>"245",
              i1=>"1", i2=>"4", ordered=>'y', value=>
              [a=>"The adventures of Huckleberry Finn /",
               c=>"Mark Twain ; illustrated by E.W. Kemble."]});

my $update245 = {field=>'245',record=>2,ordered=>'y'};
my @u245 = $x->getupdate($update245);
$x->deletemarc($update245);
$x->addfield($update245, @u245_modified);

      # outputs
$y = $x->output({'format'=>"marcmaker", charset=>\%outc});
$x->output({file=>">>my_text.txt",'format'=>"ascii",record=>2});
$x->output({file=>">my_marcmaker.mkr",'format'=>"marcmaker",
            nolinebreak=>'y',lineterm=>'\n'});
$x->output({file=>">titles.html",'format'=>"html", 245=>"Title: "});

      # manipulation of individual marc records.
```

```
        @recs = $x[1..$#$x];
        grep {$_->unpack_ldr() && 0} @recs;
        @LCs = grep {$_->unp_ldr{Desc} eq 'a' &&
                     $_->getvalue({field=>'040'}) =~/DLC\c_.DLC/} @recs;
        foreach my $rec (@LCs) {
                print $rec->output({format=>'usmarc'});
        }

              # manipulation as strings.
        foreach my $rec (@LCs) {
                my $stringvar = $rec->as_string();
                $stringvar=~s[^(
                                100\s # main entries of this stripe..
                                ..\s # (don't care about indicators)
                                \c_.\s*
                                )(\S) # take the first letter..
                              ] [
                              ${1}uc($2) # and upcase it. All authors have
                                        # upcase first letters in my library.
                              ]xm; # x means 'ignore whitespace and allow
                                   # embedded comments'.
               $rec->from_string($stringvar);
               my ($i2,$article) = $stringvar =~/245 .(.) \c_.(.{0,9})/;
               $article = substr($article,0,$i2) if $i2=~/\d/;
               print "article $article is not common" unless $COMMON_ARTS{$article};
        }
```

DESCRIPTION

MARC.pm is a Perl 5 module for reading in, manipulating, and outputting bibliographic records in the *USMARC* format. You will need to have Perl 5.004 or greater for MARC.pm to work properly. Since it is a Perl module you use MARC.pm from one of your own Perl scripts. To see what sorts of conversions are possible you can try out a web interface to MARC.pm which will allow you to upload MARC files and retrieve the results (for details see the section below entitled "Web Interface").

However, to get the full functionality you will probably want to install MARC.pm on your server or PC. MARC.pm can handle both single and batches of MARC records. The limit on the number of records in a batch is determined by the memory capacity of the machine you are running. If memory is an issue for you MARC.pm will allow you to read in records from a batch gradually. MARC.pm also includes a variety of tools for searching, removing, and even creating records from scratch.

Types of Conversions:

- MARC - ASCII : separates the MARC fields out into separate lines
- MARC <-> MARCMaker : The MARCMaker format is a format that was developed by the *Library of Congress* for use with their DOS based *MARCMaker* and *MARCBreaker* utilities. This format is particularly useful for making global changes (ie. with a text editor‘s search and replace) and then converting back to MARC (MARC.pm will read properly formatted MARCMaker records). For more information about the MARCMaker format see http://lcweb.loc.gov/marc/marcsoft.html
- MARC - HTML : The MARC to HTML conversion creates an HTML file from the fields and field labels that you supply. You could possibly use this to create HTML bibliographies from a batch of MARC records.
- MARC <-> XML : XML support is handled by MARC::XML which is a subclass of MARC.pm and is also available for download from the CPAN.
- MARC - URLS : This conversion will extract URLs from a batch of MARC records. The URLs are found in the 856 field, subfield u. The HTML page that is generated can then be used with link-checking software to determine which URLs need to be repaired. Hopefully library system vendors will soon support this activity soon and make this conversion unecessary!

Downloading and Installing

Download

The module is provided in standard CPAN distribution format. It will extract into a directory MARC-version with any necessary subdirectories. Change into the MARC top directory. Download the latest version from http://www.cpan.org/modules/by-module/MARC/

Unix

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

Win9x/WinNT/Win2000

```
perl Makefile.PL
perl test.pl
perl install.pl
```

Test

Once you have installed, you can check if Perl can find it. Change to some other directory and execute from the command line:

```
perl -e "use MARC"
```

If you do not get any response that means everything is OK! If you get an error like *Can‘t locate method "use" via package MARC.* then Perl is not able to find MARC.pm—double check that the file copied it into the right place during the install.

Todo

- Support for other MARC formats (UKMARC, FINMARC, etc).
- Create a map and instructions for using and extending the MARC.pm data structure.
- Develop better error catching mechanisms.
- Support for MARC <-> Unicode character conversions.
- MARC <-> EAD (Encoded Archival Description) conversion?
- MARC <-> DC/RDF (Dublin Core Metadata encoded in the Resource Description Framework)?

Web Interface

A web interface to MARC.pm is available at http://marcpm.sourceforge.net/cgi-bin/converter.cgi where you can upload records and observe the results. If you‘d like to check out the cgi script take a look at http://marcpm.sourceforge.net/documentation/marcpm/converter.txt However, to get the full functionality you will want to install MARC.pm on your server or PC.

Option Templates

A MARC record is a complex structure. Hence, most of the methods have a number of options. Since a series of operations frequently uses many the same options for each method, you can create a single variable that forms a "template" for the desired options. The variable points to a hash - and the hash keys have been selected so the same hash works for all of the related methods.

```
my $loc852 = {record=>1, field=>’852’, ordered=>’y’};
my ($found) = $x->searchmarc($loc852);
if (defined $found) {
    my @m852 = $x->getupdate($loc852);
    $x->deletemarc($loc852);
        # change @m852 as desired
    $x->updaterecord($loc852, @m852fix);
}
else {
    $x->addfield($loc852, @m852new);
}
```

The following methods are specifically designed to work together using *Option Templates*. The **required** options are shown as **bold**. Any `(default)` options are shown in parentheses. Although **deletemarc()** permits an array for the *record* option, a single *record* should be used in a Template. The *subfield* option must not be used in a Template that uses both **deletemarc** and one of the methods that acts on a complete *field* like **addfield()**. The *value* option must not be used with **updaterecord()**.

deletemarc()	- field (all), record (all), subfield [supplemental]
searchmarc()	- **field**, regex, notregex, subfield [supplemental]
getvalue()	- **record**, **field**, subfield, delimiter [supplemental]
getfirstvalue()	- **record**, **field**, subfield, delimiter [supplemental]
getupdate()	- **record**, **field**
getfields()	- **record**, **field**
addfield()	- **record**, **field**, i1 (' '), i2 (' '), value, ordered ('y')
updaterecord()	- **record**, **field**, i1 (' '), i2 (' '), ordered ('y')
updatefields()	- **record**, **field**
deletefirst()	- **record**, **field**, subfield
updatefirst()	- **record**, **field**, subfield, i1,i2

Methods that accept a *subfield* option also accept specifying it as a supplemental parameter. (Deletefirst and updatefirst are the only exceptions). Supplemental parameters append/overwrite the hash values specified in the template.

```
    $x->deletemarc($loc852, 'subfield','k');

    my $f260 = {field=>"260",regex=>"/19../"};
    my @records=$x->searchmarc($f260,'subfield','c');
    foreach $found (@records) {
        $value = $x->getvalue($f260,'record',"$found",'field',"245");
        print "TITLE: $value\n";
    }
```

METHODS

Here is a list of the methods in MARC.pm that are available to you for reading in, manipulating and outputting MARC data.

new()

Creates a new MARC object.

```
    $x = MARC->new();
```

You can also use the optional *file* and *format* parameters to create and populate the object with data from a file. If a file is specified it will read in the entire file. If you wish to read in only portions of the file see openmarc(), nextmarc(), and closemarc() below. The *format* defaults to `'usmarc'` if not specified. It is only used when a *file* is given.

```
    $x = MARC->new("mymarc.dat","usmarc");
    $x = MARC->new("mymarcmaker.mkr","marcmaker");
```

Creates a new MARC::Rec object.

```
    $rec=MARC::Rec->new();
    $rec=MARC::Rec->new($filehandle,"usmarc");
```

MARC::Rec objects are typically created by reading from a filehandle using nextrec() and a proto MARC::Rec object or by directly stuffing the @{$rec-{'array'}} array.

openmarc()

Opens a specified file for reading data into a MARC object. If no format is specified openmarc() will default to USMARC. The *increment* parameter defines how many records you would like to read from the file. If no *increment* is defined then the file will just be opened, and no records will be read in. If *increment* is set to -1 then the entire file will be read in.

```
    $x = new MARC;
    $x->openmarc({file=>"mymarc.dat",'format'=>"usmarc",
                  increment=>"1"});
```

```
$x->openmarc({file=>"mymarcmaker.mkr",'format'=>"marcmaker",
              increment=>"5"});
```

note: openmarc() will return the number of records read in. If the file opens successfully, but no records are read, it returns `"0 but true"`. For example:

```
$y=$x->openmarc({file=>"mymarc.dat",'format'=>"usmarc",
                 increment=>"5"});
print "Read in $y records!";
```

When the *MARCMaker* format is specified, the *lineterm* parameter can be used to override the CRLF line-ending default (the format was originally released for MS-DOS). A *charset* parameter accepts a hash-reference to a user supplied character translation table. The "usmarc.txt" table supplied with the LoC. MARCMaker utility is used internally as the default. You can use the **usmarc_default** method to get a hash-reference to it if you only want to modify a couple of characters. See example below.

```
$x->openmarc({file=>"makrbrkr.mrc",'format'=>"marcmaker",
              increment=>"5",lineterm=>"\n",
              charset=>\%char_hash});
```

openmarc inherits some error checking sanity checks from MARC::Rec::nextrec. These will lead it to return the negative of the number of records read in if there is a header length error. This behavior can be suppressed with an option:

```
$x->openmarc({file=>"mymarc.dat", format=>"usmarc",
              increment=> 1, strict => 0});
```

nextmarc()

Once a file is open nextmarc() can be used to read in the next group of records. The increment can be passed to change the number of records read in if necessary. An increment of -1 will read in the rest of the file. Specifying the increment will change the value set with openmarc(). Otherwise, that value is the default.

```
$x->nextmarc();
$x->nextmarc(10);
$x->nextmarc(-1);
```

note: Similar to openmarc(), nextmarc() will return the number of records read in.

```
$y=$x->nextmarc();
print "$y more records read in!";
```

nextrec()

MARC:Rec instances can read from a filehandle and produce a new MARC::Rec instance. If nextrec is passed a string, it will read from that instead. The string should be formatted according to the {format} field of the instance.

Cases where a new instance cannot be created are classified by a status value:

```
my ($newrec,$status) = $rec->nextrec();
```

$status is undefined if we are at the end of the filehandle. If the data read from the filehandle cannot be made into a marc record, $status will be negative. For example, $status is -1 if there is a distinction between recsize and leader definition of recsize, and -2 if the leader is not numeric.

An idiom for reading records incrementally with MARC::Recs is:

```
my $proto=MARC::Rec->new($filehandle,$format);
while (1) {
      my ($rec,$status)=$proto->nextrec();
      last unless $status;
      die "Bad record, bad, bad record: error $status"
          if $status <0;
      print $rec->output({$format=>'ascii'});
      # or replace print and output with your own functions/methods.
}
close $filehandle or die "File $filehandle is not happy on close\n";
```

If you are getting records from an external source as strings, the idiom is:

```
    my $proto=MARC::Rec->new($filehandle,$format);
    while (1) {
          my $string = get_external_marc();
          last unless $string;
          my ($rec,$status)=$proto->nextrec($string);
          last unless $status;
          die "Bad record, bad, bad record: error $status"
              if $status <0;
          print $rec->output({$format=>'ascii'});
          # or replace print and output with your own functions/methods.
    }
```

closemarc()

If you are finished reading in records from a file you should close it immediately.

```
    $x->closemarc();
```

add_map()

add_map() takes a recnum and a ref to a field in ($tag, $i1,$i2,a="bar",...) or ($tag, $field) formats and will append to the various indices that we have hanging off that record. It is intended for use in creating records de novo and as a component for rebuild_map(). It carefully does not copy subfield values or entire fields, maintaining some reference relationships. What this means for indices created with add_map that you can directly edit subfield values in $marc-[recnum]{array} and the index will adjust automatically. Vice-versa, if you edit subfield values in $marc-{recnum}{tag}{subfield_code} the fields in $marc-[recnum]{array} will adjust. If you change structural information in the array with such an index, you must rebuild the part of the index related to the current tag (and possibly the old tag if you change the tag).

```
   use MARC 1.02;
   while (<>) {
        chomp;
        my ($author,$title) = split(/\t/);
        my $rnum = $x->createrecord({leader=>
                                        "00000nmm  2200000 a 4500"});

        my @auth = (100, ' ', ' ', a=>$author);
        my @title = (245, ' ', ' ', a=>$title);
        push @{$x->[$rnum]{array}}, \@auth;
        $x->add_map($rnum,\@auth);
        push @{$x->[$rnum]{array}}, \@title;
        $x->add_map($rnum,\@title);
   }
```

MARC::Rec::add_map($rfield) does not need the record specification and has the same effect as add_map.

rebuild_map

rebuild_map takes a recnum and a tag and will synchronize the index with the array elements of the marc record at the recnum with that tag.

```
      #Gonna change all 099's to 092's since this is a music collection.
      grep {$->[0] =~s/099/092} @{$x->[$recnum]{array}};

      #Oops, now the index is out of date on the 099's...
      $x->rebuild_map($recnum,099);
      #... and the 092's since we now have new ones.
      $x->rebuild_map($recnum,092);
      #All fixed.
```

MARC::Rec::rebuild_map($tag) does not need the record number and has the same effect as rebuild_map.

rebuild_map_all

rebuild_map takes a recnum and will synchronize the index with the array elements of the marc record at the recnum.

MARC::Rec::rebuild_map_all() does not need the record number and has the same effect as rebuild_map_all.

getfields

getfields takes a template and returns an array of fieldrefs from the record number implied by that template. The fields referred are fields from the $marc-[$recnum]{array} group. The fields are all fields from the first one with the tag from the template to the last with that tag. Some marc records (e.g. cjk) may have fields with other tags mixed in. Consecutive calls to updatefields with a different tag and the same record are probably a bad idea unless you have assurance that fields with the same tag are always together.

MARC::Rec::getfields is identical to getfields, but ignores any record specification in the template.

marc_count()

Returns the total number of records in a MARC object. This method was previously named **length()**, but that conflicts with the Perl built-in of the same name. Use the new name, the old one is deprecated and will disappear shortly.

```
    $length=$x->marc_count();
```

getfirstvalue()

getfirstvalue will return the first value of a field or subfield or indicator or i12 in a particular record found in the MARC object. It does not depend on the index being up to date.

MARC::Rec::getfirstvalue is identical to getfirstvalue, but ignores any record specification in the template.

getvalue()

This method will retrieve MARC field data from a specific record in the MARC object. getvalue() takes four parameters: *record*, *field*, *subfield*, and *delimiter*. Since a single MARC record could contain several of the fields or subfields the results are returned to you as an array. If you only pass *record* and *field* you will be returned the entire field without subfield delimiters. Optionally you can use *delimiter* to specify what character to use for the delimiter, and you will also get the subfield delimiters. If you also specify *subfield* your results will be limited to just the contents of that subfield. Repeated subfield occurances will end up in separate array elements in the order in which they were read in. The *subfield* designations `‘i1‘, ‘i2’ and ‘i12’` can be used to get indicator(s).

```
        #get the 650 field(s)
    @results = $x->getvalue({record=>’1’,field=>’650’});

        #get the 650 field(s) with subfield delimiters (ie. |x |v etc)
    @results = $x->getvalue({record=>’1’,field=>’650’,delimiter=>’|’});

        #get all of the subfield u’s from the 856 field
    @results = $x->getvalue({record=>’12’,field=>’856’,subfield=>’u’});
```

MARC::Rec::getvalue($template) is identical to getvalue, but ignores any record specification.

unpack_ldr($record)

Returns a ref to a hash version of the record‘th LDR. Installs the ref in $marc as
$marc-[$record]{unp_ldr}

```
    my $rldr = $x->unpack_ldr(1);
    print "Desc is $rldr{Desc}";
    my ($m040) = $x->getvalues({record=>’1’,field=>’040’});
    print "First record is LC, let’s leave it alone"
          if $rldr->{’Desc’} eq ’a’ && $m040=~/DLC\s*\c_c\s*DLC/;
```

The hash version contains the following information:

```
        Key             000-Pos length  Function [standard value]
        ---             ------- ------  --------
        rec_len         00-04      5    Logical Record Length
        RecStat         05         1    Record Status
        Type            06         1    Type of Record
```

```
    BLvl            07         1    Bibliographic Level
    Ctrl            08         1
    Undefldr        09-11      3    [x22]
    base_addr       12-16      5    Base Address of Data
    ELvl            17         1    Encoding Level
    Desc            18         1    Descriptive Cataloging Form
    ln_rec          19         1    Linked-Record Code
    len_len_field   20         1    Length "length of field" [4]
    len_start_char  21         1    Length "start char pos" [5]
    len_impl        22         1    Length "implementation dep" [0]
    Undef2ldr       23         1    [0]
```

MARC::Rec::unpack_ldr() is identical to unpack_ldr, but does not need the record number.

get_hash_ldr($record)

Takes a record number. Returns a ref to the cached version of the hash ldr if it exists. Does this *without* overwriting the hash ldr. Allows external code to safely manipulate hash versions of the ldr.

```
  my $rhldr = $marc->get_hash_ldr($record);
  return undef unless $rhldr;
  $rhldr->{'Desc'} =~ s/a/b/;
  $ldr = $x->pack_ldr($record);
```

MARC::Rec::get_hash_ldr() is identical to get_hash_ldr, but does not need the record number.

pack_ldr($record)

Takes a record number. Updates the appropriate ldr.

```
  $marc->[$record]{'unp_ldr'}{'Desc'} =~ s/a/b/;
  my $ldr = $x->pack_ldr($record);
  return undef unless $ldr;
```

MARC::Rec::pack_ldr() is identical to pack_ldr, but does not need the record number.

bib_format($record)

Takes a record number. Returns the "format" used in determining the meanings of the fixed fields in 008. Will force update of the ldr based on any existing hash version.

```
   foreach $record (1..$#$x) {
         next if $x->bib_format($record) eq 'SERIALS';
             # serials are hard
         do_something($x->[record]);
   }
```

MARC::Rec::bib_format() is identical to bib_format, but does not need the record number.

unpack_008($record)

Returns a ref to hash version of the 008 field, based on the field‘s value. Installs the ref as $marc-[$record]{unp_008}

```
   foreach $record (1..$#$x) {
         my $rff = $x->unpack_008($record);
         print "Record $record: Y2K problem possible"
             if ($rff->{'Date1'}=~/00/ or $rff->{'Date2'}=~/00/);
   }
```

MARC::Rec::unpack_008() is identical to unpack_008, but does not need the record number.

get_hash_008($record)

Takes a record number. Returns a ref to the cached version of the hash 008 if it exists. Does this *without* overwriting the hash 008. Allows external code to safely manipulate hash versions of the 008.

```
  my $rh008 = $marc->get_hash_008($record);
  return undef unless $rh008;
  $rh008->{'Date1'} =~ s/00/01/;
```

```
    my $m008 = $x->pack_008($record);
    return undef unless $m008;
```

MARC::Rec::get_hash_008() is identical to get_hash_008, but does not need the record number.

pack_008($record)

Takes a record number and updates the appropriate 008. Will force update of the ldr based on any existing hash version. Updates the map for 008.

```
    foreach $record (1..$#$x) {
          my $rff = $x->unpack_008($record);
          $rff->{’Date1’}=’2000’;
          print "Record:$record Y2K problem created\n";
          $x->pack_008($record);
          # New value is in the 008 field of $record’th marc
    }
```

MARC::Rec::pack_008() is identical to pack_008, but does not need the record number.

deletefirst()

deletefirst() takes a template. It deletes the field data for a first match, using the template and leaves the rest alone.
For example, assume that we have a marc file whose second record looks like:

```
    =008  960221s1955\\\\dcuabcdjdbkoqu001\0dspa\d
    =020  \\$a0777000008 :$c{24}35.99
    =020  \\$a0777000008 :$c{curren}35.99
    =040  \\$aViArRB$cViArRB
    =100  2 $aDeer-Doe, Jane,$d1957-
```

Assume we have placed this in a MARC object x. Then we can delete an entire field of the second record thus:

```
    my $loc100 = {record=>2,field=>100,rebuild_map=>1};
    $x->deletefirst($loc100);
```

The second record now looks like:

```
    =008  960221s1955\\\\dcuabcdjdbkoqu001\0dspa\d
    =020  \\$a0777000008 :$c{24}35.99
    =020  \\$a0777000008 :$c{curren}35.99
    =040  \\$aViArRB$cViArRB
```

If the template has a subfield element it deletes based on the subfield information in the template.

```
    $x->deletefirst({record=>2,field=>020,subfield=>’c’,rebuild_map=>1};

    =008  960221s1955\\\\dcuabcdjdbkoqu001\0dspa\d
    =020  \\$a0777000008 :
    =020  \\$a0777000008 :$c{curren}35.99
    =040  \\$aViArRB$cViArRB
```

If the last subfield of a field is deleted, deletefirst() also deletes the field.

```
    $x->deletefirst({record=>2,field=>020,subfield=>’a’,rebuild_map=>1};

    =008  960221s1955\\\\dcuabcdjdbkoqu001\0dspa\d
    =020  \\$a0777000008 :$c{curren}35.99
    =040  \\$aViArRB$cViArRB
```

It complains about attempts to delete indicators.

```
    $x->deletefirst({record=>2,field=>020,subfield=>’i1’,rebuild_map=>1};
    OR
    $x->deletefirst({record=>2,field=>020,i1=>1,rebuild_map=>1};
```

Produces a warning and leaves the record untouched.

```
 If there is no match, it does nothing.

    $x->deletefirst({record=>2,field=>020,subfield=>'x',rebuild_map=>1};

    =008  960221s1955\\\\dcuabcdjdbkoqu001\0dspa\d
    =020  \\$a0777000008 :$c{curren}35.99
    =040  \\$aViArRB$cViArRB
```

Deletefirst also rebuilds the map if the template asks for that $do_rebuild_map. Deletefirst returns the number of matches deleted (that would be 0 or 1), or undef if it feels grumpy (i.e. carps).

MARC::Rec::deletefirst($template) is identical to deletefirst, but ignores any record number specified by $template.

Most use of deletefirst is expected to be by Tie::MARC.

deletemarc()

This method will allow you to remove a specific record, fields or subfields from a MARC object. Accepted parameters include: *record*, *field* and *subfield*. Note: you can use the .. operator to delete a range of records. deletemarc() will return the number of items deleted (be they records, fields or subfields). The *record* parameter is optional. It defaults to all user records [1..$#marc] if not specified.

```
        #delete all the records in the object
    $x->deletemarc();

        #delete records 1-5 and 7
    $x->deletemarc({record=>[1..5,7]});

        #delete all of the 650 fields from all of the records
    $x->deletemarc({field=>'650'});

        #delete the 110 field in record 2
    $x->deletemarc({record=>'2',field=>'110'});

        #delete all of the subfield h's in the 245 fields
    $x->deletemarc({field=>'245',subfield=>'h'});
```

updatefirst()

updatefirst() takes a template, and an array from $marc-[recnum]{array}. It replaces/creates the field data for a first match, using the template and the array, and leaves the rest alone. If the template has a subfield element, (this includes indicators) it ignores all other information in the array and only updates/creates based on the subfield information in the array. If the template has no subfield information then indicators are left untouched unless a new field needs to be created, in which case they are left blank.

MARC::Rec::updatefirst($template) is identical to deletefirst, but ignores any record number specified by $template.

Most use of updatefirst() is expected to be from Tie::MARC. It does not currently provide a useful return value.

updatefields()

updatefields() takes a template which specifies recnum, a $do_rebuild_map and a field (needs the field in case $rafields-[0] is empty). It also takes a ref to an array of fieldrefs formatted like the output of getfields(), and replaces/creates the field data. It assumes that it should replace the fields with the first tag in the fieldrefs. It calls rebuild_map() if $do_rebuild_map.

```
    #Let's kill the *last* 500 field.
    my $loc500 = {record=>1,field=>500,rebuild_map=>1};
    my @rfields = $x->getfields($loc500);
    pop @rfields;
    $x->updatefields($loc500,\@rfields);
```

getmatch()

getmatch() takes a subfield code (can be an indicator) and a fieldref. Returns 0 or a ref to the value to be updated.

```
    #Let’s update the value of i2 for the *last* 500
    my $loc500 = {record=>1,field=>500,rebuild_map=>1};
    my @rfields = $x->getfields($loc500);
    my $rvictim = pop @rfields;
    my $rval = getmatch(’i2’,$rvictim);
    $$rval = "4" if $rval;
```

MARC::Rec::getmatch($subf,$rfield) is identical to getmatch.

insertpos()

insertpos() takes a subfield code (can not be an indicator), a value, and a fieldref. Updates the fieldref with the first place that the fieldref can match. Assumes there is no exact subfield match in $fieldref.

```
    #Let’s update the value of subfield ’a’ for the *last* 500
    my $value = "new info";
    my $loc500 = {record=>1,field=>500,rebuild_map=>1};
    my @rfields = $x->getfields($loc500);
    my $rvictim = pop @rfields;
    my $rval = getmatch(’a’,$rvictim);
    if ($rval) {
        $$rval = $value ;
    } else {
        $x->insertpos(’a’,$value,$rvictim);
    }
```

MARC::Rec::insertpos($subf,$value,$rfield) is identical to insertpos;

selectmarc()

This method will select specific records from a MARC object and delete the rest. You can specify both individual records and ranges of records in the same way as deletemarc(). selectmarc() will also return the number of records deleted.

```
    $x->selectmarc([’3’]);
    $y=$x->selectmarc([’4’,’21-50’,’60’]);
    print "$y records selected!";
```

searchmarc()

This method will allow you to search through a MARC object, and retrieve record numbers for records that matched your criteria. You can search for: 1) records that contain a particular field, or field and subfield ; 2) records that have fields or subfields that match a regular expression ; 3) and records that have fields or subfields that **do not** match a regular expression. The record numbers are returned to you in an array which you can then use with deletemarc(), selectmarc() and output() if you want.

- 1) Field/Subfield Presence:

```
    @records=$x->searchmarc({field=>"245"});
    @records=$x->searchmarc({field=>"245",subfield=>"a"});
```

- 2) Field/Subfield Match:

```
    @records=$x->searchmarc({field=>"245",
                             regex=>"/huckleberry/i"});
    @records=$x->searchmarc({field=>"260",subfield=>"c",
                             regex=>"/19../"});
```

- 3) Field/Subfield NotMatch:

```
    @records=$x->searchmarc({field=>"245",
                             notregex=>"/huckleberry/i"});
    @records=$x->searchmarc({field=>"260",
                             subfield=>"c",notregex=>"/19../"});
```

createrecord()

You can use this method to initialize a new record. It only takes one optional parameter, *leader* which sets the 24 characters in the record leader: see http://lcweb.loc.gov/marc/bibliographic/ecbdhome.html for more details on the leader. Note: you do not need to pass character positions 00-04 or 12-16 since these are calculated by MARC.pm if outputting to MARC you can assign 0 to each position. If no leader is passed a default USMARC leader will be created of "00000nam 2200000 a 4500". createrecord() will return the record number for the record that was created, which you will need to use later when adding fields with addfield(). Createrecord now makes the new record an instance of an appropriate MARC::Rec subclass.

```
use MARC;
my $x = new MARC;
$record_number = $x->createrecord();
$record_number = $x->createrecord({leader=>
                                   "00000nmm  2200000 a 4500"});
```

MARC::Rec::createrecord($leader) returns an instance of a suitable subclass of MARC::Rec.

getupdate()

The **getupdate()** method returns an array that contains the contents of a field in a defined order that permits restoring the field after deleting it. This permits changing only individual subfields while keeping other data intact. If a field is repeated in the record, the resulting array separates the field information with an element containing "\036" - the internal field separator which can never occur in real MARC data parameters. A non-existing field returns `undef`. An example will make the structure clearer. The next two MARC fields (shown in ASCII) will be described in the following array:

```
            246  30  $aPhoto archive
            246  3   $aAssociated Press photo archive

my $update246 = {field=>'246',record=>2,ordered=>'y'};
    # next two statements are equivalent
my @u246 = $x->getupdate($update246);
    # or
my @u246 = ('i1','3','i2','0',
            'a','Photo archive',"\036",
            'i1','3','i2',' ',
            'a','Associated Press photo archive',"\036");
```

After making any desired modifications to the data, the existing field can be replaced using the following sequence (for non-repeating fields):

```
$x->deletemarc($update246));
my @records = ();
foreach my $y1 (@u246) {
    last if ($y1 eq "\036");
    push @records, $y1;
}
$x->addfield($update246, @records);
```

updaterecord()

The updaterecord() method is a more complete version of the preceding sequence with error checking and the ability to split the update array into multiple addfield() commands when given repeating fields. It takes an array of key/value pairs, formatted like the output of getupdate(), and replaces/creates the field data. For repeated tags, a "\036" element is used to delimit data into separate addfield() commands. It returns the number of successful addfield() commands or `undef` on failure.

```
$repeats = $x->updaterecord($update246, @u246);     # same as above
```

addfield()

This method will allow you to addfields to a specified record. The syntax may look confusing at first, but once you understand it you will be able to add fields to records that you have read in, or to records that you have created with createrecord(). addfield() takes six parameters: *record* which indicates the record number to add the field to, *field* which indicates the field you wish to create (ie. 245), *i1* which holds one character for the first indicator, *i2* which holds one character for the second indicator, and *value* which holds the subfield data that you wish to add to the field. addfield() will automatically try to insert your new field in tag order (ie. a 500 field before a 520 field), however you can turn this off if you set *ordered* to "no" which will add the field to the end. Here are some examples:

```
$y = $x->createrecord(); # $y will store the record number created

$x->addfield({record=>"$y", field=>"100", i1=>"1", i2=>"0",
              value=> [a=>"Twain, Mark, ", d=>"1835-1910."]});

$x->addfield({record=>"$y", field=>"245",
              i1=>"1", i2=>"4", value=>
             [a=>"The adventures of Huckleberry Finn /",
              c=>"Mark Twain ; illustrated by E.W. Kemble."]});
```

This example initialized a new record, and added a 100 field and a 245 field. For some more creative uses of the addfield() function take a look at the *EXAMPLES* section. The *value* parameters, including *i1* and *i2*, can be specified using a separate array. This permits restoring field(s) from the array returned by the **getupdate()** method - either as-is or with modifications. The *i1* and *i2* key/value pairs must be first and in that order if included.

```
    # same as "100" example above
my @v100 = ’i1’,’1’,’i2’,"0",’a’,"Twain, Mark, ",
           ’d’,"1835-1910.";
$x->addfield({record=>"$y", field=>"100"}, @v100);
```

add_005s()

Add_005s takes a specification of records (defaults to everything) and updates the indicated records with updated 005 fields (date of last transaction).

output()

Output is a multi-functional method for creating formatted output from a MARC object. There are three parameters *file*, *format*, *records*. If *file* is specified the output will be directed to that file. It is important to specify with and whether you want to create or append the file! If no *file* is specified then the results of the output will be returned to a variable (both variations are listed below).

The MARC standard includes a control field (005) that records the date of last automatic processing. This is implemented as a side-effect of output() to a file or if explicitly requested via a add_005s field of the template. The current time is stamped on the records indicated by the template.

Valid *format* values currently include usmarc, marcmaker, ascii, html, urls, and isbd. The optional *records* parameter allows you to pass an array of record numbers which you wish to output. You must pass the array as a reference, hence the forward-slash in \@records below. If you do not include *records* the output will default to all the records in the object.

The *lineterm* parameter can be used to override the line-ending default for any of the formats. *MARCMaker* defaults to CRLF (the format was originally released for MS-DOS). The others use ‘\n’ as the default.

With the *MARCMaker* format, a *charset* parameter accepts a hash-reference to a user supplied character translation table. The "ustext.txt" table supplied with the LoC. MARCBreaker utility is used internally as the default. You can use the **ustext_default** method to get a hash-reference to it if you only want to modify a couple of characters. See example below.

The *MARCMaker* Specification requires that long lines be split to less than 80 columns. While that behavior is the default, the *nolinebreak* parameter can override it and the resulting output will be much like the *ascii* format.

MARC::Rec::output($template) is the same as output except that ignores record number(s) and only outputs its caller. (E.g., with $format eq ‘urls’ it does not output html header and footer information.)

- MARC

  ```
  $x->output({file=>">mymarc.dat",’format’=>"usmarc"});
  $x->output({file=>">mymarc.dat",’format’=>"usmarc",
              records=>\@records});
  $y=$x->output({’format’=>"usmarc"}); #put the output into $y
  ```

- MARCMaker

  ```
  $x->output({file=>">mymarcmaker.mkr",’format’=>"marcmaker"});
  $x->output({file=>">mymarcmaker.mkr",’format’=>"marcmaker",
              records=>\@records});
  ```

```
    $y=$x->output({'format'=>"marcmaker"}); #put the output into $y

    $x->output({file=>"brkrtest.mkr",'format'=>"marcmaker",
                nolinebreak=>"1", lineterm=>"\n",
                charset=>\%char_hash});
```

- ASCII
```
    $x->output({file=>">myascii.txt",'format'=>"ascii"});
    $x->output({file=>">myascii.txt",'format'=>"ascii",
                records=>\@records});
    $y=$x->output({'format'=>"ascii"}); #put the output into $y
```

- HTML
The HTML output method has some additional parameters. *fields* which if set to "all" will output all of the fields. Or you can pass the tag number and a label that you want to use for that tag. This will result in HTML output that only contains the specified tags, and will use the label in place of the MARC code.
```
    $x->output({file=>">myhtml.html",'format'=>"html",
                fields=>"all"});

        #this will only output the 100 and 245 fields, with the
        #labels "Title: " and "Author: "
    $x->output({file=>">myhtml.html",'format'=>"html",
                245=>"Title: ",100=>"Author: "});

    $y=$x->output({'format'=>"html"});
```
If you want to build the HTML file in stages, there are four other *format* values available to you: 1) "html_header", 2) "html_start", 3) "html_body", and 4) "html_footer". Be careful to use the append when adding to a file though!
```
    $x->output({file=>">myhtml.html",
                'format'=>"html_header"}); # Content-type
    $x->output({file=>">>myhtml.html",
                'format'=>"html_start"});  # <BODY>
    $x->output({file=>">>myhtml.html",
                'format'=>"html_body",fields=>"all"});
    $x->output({file=>">>myhtml.html",
                'format'=>"html_footer"});
```

- URLS
```
    $x->output({file=>"urls.html",'format'=>"urls"});
    $y=$x->output({'format'=>"urls"});
```

- ISBD
An experimental output format that attempts to mimic the ISBD.
```
    $x->output({file=>"isbd.txt",'format'=>"isbd"});
    $y=$x->output({'format'=>"isbd"});
```

- XML
Round-trip conversion between MARC and XML is handled by the subclass MARC::XML. MARC::XML is available for download from the CPAN.

usmarc_default()

This method returns a hash reference to a translation table between mnemonics delimited by curly braces and single-byte character codes in the MARC record. Multi-byte characters are not currently supported. The hash has keys of the form '{esc}' and values of the form chr(0x1b). It is used during MARCMaker input.
```
my %inc = %{$x->usmarc_default()};
printf "dollar = %s\n", $inc{'dollar'};     # prints '$'
$inc{'yen'} = 'Y';
```

```
    $x->openmarc({file=>"makrbrkr.mrc",'format'=>"marcmaker",
                  charset=>\%inc});
```

MARC::Rec::usmarc_default is identical to usmarc_default;

ustext_default()

This method returns a hash reference to a translation table between single-byte character codes and mnemonics delimited by curly braces. Multi-byte characters are not currently supported. The hash has keys of the form chr(0x1b) and values of the form '{esc}'. It is used during MARCMaker output.

```
    my %outc = %{$x->ustext_default()};
    printf "dollar = %s\n", $outc{'$'}; # prints '{dollar}'
    $outc{'$'} = '{uscash}';
    printf "dollar = %s\n", $outc{'$'}; # prints '{uscash}'
    $y = $x->output({'format'=>"marcmaker", charset=>\%outc});
```

MARC::Rec::ustext_default is identical to ustext_default;

as_string()

As_string() takes no parameters and returns a (Unix) newline separated version of the record.

```
  Format is: $tag<SPACE>$i1$i2<SPACE>$subfields
  where $subfields are separated by "\c_" binary subfield indicators.
  Tag 000 is ldr.
```

Subclasses may need to override this format. If so, they should override from_string.

from_string()

From_string() takes a string parameter and updates the calling record's {array} information. It assumes the string is formatted like the output of as_string().

EXAMPLES

Here are a few examples to fire your imagination.

- This example will read in the complete contents of a MARC file called "mymarc.dat" and then output it as a MARCMaker file called "mymkr.mkr".

```
    #!/usr/bin/perl
    use MARC;
    $x = MARC->new("mymarc.dat","marcmaker");
    $x->output({file=>"mymkr.mkr",'format'=>"marcmaker");
```

- The MARC object occupies a fair number of working memory, and you may want to do conversions on very large files. In this case you will want to use the openmarc(), nextmarc(), deletemarc(), and closemarc() methods to read in portions of the MARC file, do something with the record(s), remove them from the object, and then read in the next record(s). This example will read in one record at a time from a MARC file called "mymarc.dat" and convert it to a MARC Maker file called "myfile.mkr".

```
    #!/usr/bin/perl
    use MARC;
    $x = new MARC;
    $x->openmarc({file=>"mymarc.dat",'format'=>"usmarc"});
    while ($x->nextmarc(1)) {
        $x->output({file=>">>myfile.mkr",'format'=>"marcmaker"});
        $x->deletemarc(); #empty the object for reading in another
    }
```

- Perhaps you have a tab delimited text file of data for online journals you have access to from Dow Jones Interactive, and you would like to create a batch of MARC records to load into your catalog. In this case you can use createrecord(), addfield() and output() to create records as you read in your delimited file. When you are done, you then output to a file in USMARC.

```
    #!/usr/bin/perl
    use MARC;
    $x = new MARC;
    open (INPUT_FILE, "delimited_file");
    while ($line=<INPUT_FILE>) {
        ($journaltitle,$issn) = split /\t/,$line;
        $num=$x->createrecord();
        $x->addfield({record=>$num,
                      field=>"022",
                      i1=>" ", i2=>" ",
                      value=>$issn});
        $x->addfield({record=>$num,
                      field=>"245",
                      i1=>"0", i2=>" ",
                      value=>[a=>$journaltitle]});
        $x->addfield({record=>$num,
                      field=>"260",
                      i1=>" ", i2=>" ",
                      value=>[a=>"New York (N.Y.) :",
                              b=>"Dow Jones & Company"]});
        $x->addfield({record=>$num,
                      field=>"710",
                      i1=>"2", i2=>" ",
                      value=>[a=>"Dow Jones Interactive."]});
        $x->addfield({record=>$num,
                      field=>"856",
                      i1=>"4", i2=>" ",
                      value=>[u=>"http://www.djnr.com",
                              z=>"Connect"]});
    }
    close INPUT_FILE;
    $x->output({file=>">dowjones.mrc",'format'=>"usmarc"})
```

- Perhaps you have periodicals coming in that you want to order by location and then title. MARC::Rec‘s get you out of some array indexing.

```
    #!/usr/bin//perl
    use MARC 1.03;

    my @newmarcs=@$marc[1..$#$marc]; # array slice.
    my @sortmarcs = sort by_loc_oclc @newmarcs;
    @marc[1..$#$marc] = @sortmarcs;

    sub by_loc_title {
        my ($aloc,$atitle) = loc_title($a);
        my ($bloc,$btitle) = loc_title($b);
        return  $aloc cmp $bloc
                      ||
              $atitle cmp $btitle;
    }

    sub loc_title {
        my ($rec)=@_;
        my $n049 = $rec->getfirstvalue({field=>040});
        my ($loc) = $n049=~/(ND\S+)/; # Or the first two letters of your OCLC
                                      # location.

        my $title = $rec->getfirstvalue({field=>100,delimiter=>" "});

        return ($loc,$title);
    }
```

NOTES

Please let us know if you run into any difficulties using MARC.pm—we‘d be happy to try to help. Also, please contact us if you notice any bugs, or if you would like to suggest an improvement/enhancement. Email addresses are listed at the bottom of this page.

Development of MARC.pm and other library oriented Perl utilities is conducted on the Perl4Lib listserv. Perl4Lib is an open list and is an ideal place to ask questions about MARC.pm. Subscription information is available at http://www.vims.edu/perl4lib

Two global boolean variables are reserved for test and debugging. Both are "0" (off) by default. The `$TEST` variable disables internal error messages generated using *Carp*. It also overrides the date_stamp in the "005" field with a constant "19960221075055.7". It should only be used in the automatic test suite. The `$DEBUG` variable adds verbose diagnostic messages. Since both variables are used only in testing, *MARC::Rec* uses `$MARC::TEST` and `$MARC::DEBUG` rather than define a second pair.

AUTHORS

Chuck Bearden cbearden@rice.edu
Bill Birthisel wcbirthisel@alum.mit.edu
Derek Lane dereklane@pobox.com
Charles McFadden chuck@vims.edu
Ed Summers ed@cheetahmail.com

SEE ALSO

perl(1), http://lcweb.loc.gov/marc

COPYRIGHT