Writing Cleaner Perl and Sybperl Code

By Michael Peppler

The creator of sybperl (the Sybase extensions to perl) provides new techniques for using one of the most popular system administration and CGI programming tools available.

Sybperl is the generic name for the Sybase extensions to perl, the Practical Extraction and Reporting Language, developed by Larry Wall. Perl is used for system administration, CGI programming, and has also found its way into more general applications. Although perl was initially a Unix tool, it is now available on Windows NT and Windows 95.

The sybperl package includes perl versions of both the DB-library and Client Library APIs, which makes perl one of the Sybase user's best friends. In this article, I focus on the Sybase::CTlib module (which, as the name implies, is based on Sybase's ClientLibrary), as this is the API that Sybase encourages us to use, to illustrate new ways to write cleaner, more maintainable perl and sybperl code. This follows up on the introduction to Sybase::CTlib published in the Spring 1996 issue of the ISUG magazine. The text of that article is also available on the ISUG website or on my web page at www.mbay.net/~mpeppler/SybTools–1.00.tar.gz.

Sybperl is not the only way to access a Sybase database from perl. There is also a Sybase module that conforms to the DBI specification—see page 8.

Introduction

Perl is a flexible language. Variables need not be declared: They can convert from string to number and back depending on context; strings can be turned into subroutine or variable names; and all sorts of other tricks and shortcuts can be used. Arrays are created dynamically, and can grow or shrink as usage demands it—the limit usually being the amount of memory your program can access.

Associative arrays (referred to as "hashes" by perl users) are another very powerful feature. These are arrays where the index (or key) can be a number or a string: (for example we could write `$eyes{'bob'} 'blue'` or `$color = $eyes{'jane'}`). I use hashes with database code, returning data where the column name is the key (for example, assuming a table with columns "userName" and "userID", we could have `$data{'userName'}` and `$data{'userID'}`.) This flexibility gives the programmer a lot of power, and enables us to write code very quickly (I estimate that it takes me about ten times longer to write an equivalent program in C than in perl).

There is a downside to this flexibility, however. In larger scripts it can become difficult to remember which variables have been used, and typos and other obscure errors can become difficult to find and correct. In addition, although Client Library programming in perl is easier than in C, it still requires some specific knowledge and training.

Warnings and "strict" Settings

The first thing to always do is to turn on verbose warnings. You do this by adding the -w switch to the perl command line (on a Unix system, this would mean putting `#!/usr/local/bin/perl -w` at the top of your scripts, assuming the perl binary is located in the /usr/local/bin directory).

Michael Peppler is a senior consultant with Data Migrations, Inc. He has worked with Sybase products since 1989. He wrote the sybperl prototype in 1990 and has continued to enhance it extensively. He can be reached at mpeppler@mbay.net.

Turning on verbose warnings will point out potential errors and unsafe constructs. Your scripts should always run cleanly with warnings turned on. (Note: Sybase::CTlib returns NULL values as the perl special value undef. When warnings are turned on, accessing an undefined value causes a warning. This is an annoying but necessary consequence of setting verbose warnings.)

The second thing to do is to enable the "strict" pragma. This is done by adding

```
use strict;
```

near the top of your scripts. When "strict" is in effect you have to either explicitly declare all your local variables, or prefix them with the package name (so global variable $foo would have to be written $main::foo, for example). Again, although this adds some work for the programmer, it will ultimately simplify the job by avoiding using incorrect variable names, and it will also avoid using strings where variables or subroutine calls where intended.

Turning on -w warnings and the "strict" pragma will flag a lot of potentially incorrect code. It will force you to use local variables ("my" variables) in subroutines, and will help avoid hours of frustrating debugging due to some obscure spelling error somewhere in your code.

I *always* use strict and verbose warnings, even when I write a ten-line throw-away script! It has become automatic for me to start writing with:

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

use strict;

etc...
```

This has saved me from making stupid mistakes any number of times by pointing out potentially incorrect syntax.

Grouping the SQL Code

When we write a perl script we all tend to write the SQL queries in line (i.e., put the SQL code, and the perl code needed to do the query right where it's being called). This is OK for small scripts, but can turn into quite a liability when the program becomes part of a larger set (for example, a CGI script when it is part of a larger application) or if the program itself becomes large. In that case, it is much better to move all the SQL to one module, where any changes to the schema of the underlying database can be applied easily without having to sift through all the source code to find references to the SQL that needs to be changed. So if, for example, you have a query where you retrieve user data based on the user ID, you would create a subroutine in the data access module called getUserByID() which might look something like:

```
sub getUserByID {
    my ($dbh, $userID) = @_;
    my ($restype, @row, @data);

    $dbh->ct_execute("select * from users where userID =
       $userID");
    while($dbh->ct_results($restype) == CS_SUCCEED) {
        next unless $dbh->ct_fetchable($restype);
        while(@data = $dbh->ct_fetch) {
            @row = @data;
        }
    }
    @row;
}
```

Of course, you may also have decided that all access to the database data should go through stored procedures, which is also a very good technique to limit the impact of potential schema changes from application code. Using stored procedures for all database access does not preclude any of the methods or ideas presented in this article.

Specialized Data Query Routines

Another very useful technique is to write higher level calls that retrieve data from the database in a pre-formatted way. For example, let's say you need to retrieve a single value from a table (something like select max(date) from the_table). Normally you would write something like:

```
$dbh->ct_execute("select max(date) from the_table");
while($dbh->ct_results($restype) == CS_SUCCEED) {
    next unless $dbh->ct_fetchable($restype);
    while(@data = $dbh->ct_fetch) {
        $date = $data[0];
    }
}
```

When the loop is exited, the variable $date holds the result of the query. This will work perfectly well, of course, but if you need to do this sort of query more than a couple of times in your program, it becomes tedious to write the fetch loop each time. It might be better to write a specialized subroutine to do this:

```
sub Scalar {
        my $dbh = shift;
        my $sql = shift;

        my $restype;
        my @data;

        $dbh->ct_execute($sql) == CS_SUCCEED or return undef;

        while($dbh->ct_results($restype) == CS_SUCCEED) {
                next unless($dbh->ct_fetchable($restype));
                @data = $dbh->ct_fetch();
                # we're only interested in the first row, so cancel
                # any further results
                $dbh->ct_cancel(CS_CANCEL_ALL);
        }

        # return the first column of the row
        $data[0];
}
```

Now the query would be written:

```
$date = Scalar($dbh, "select max(date) from the_table");
```

In the same vein, there are quite often queries that we know return just one row. In that case we could use a "HashRow" subroutine:

```
sub HashRow {
        my $dbh = shift;
        my $sql = shift;
        my $restype;
        my %data;

        $dbh->ct_execute($sql) == CS_SUCCEED or return undef;

        while($dbh->ct_results($restype) == CS_SUCCEED) {
                next unless $dbh->ct_fetchable($restype);
                # fetch one row as a hash (associative array)
                %data = $dbh->ct_fetch(CS_TRUE);
                # ignore any further results
                $dbh->ct_cancel(CS_CANCEL_ALL);
        }

        # return a reference to this hash:
        { %data };
}
```

So, if you need to get a single row you can do something like:

```
$data = HashRow($dbh, "select * from sysusers where uid = 101");
```

And you can access various columns of the row that is returned like this:

```
$user = $data->{'user'};
```

As you can imagine this technique can be extended to allow all sorts of specialized queries, or to return data that is formatted in a special way.

For example, I recently improved the speed of a http log file processing program by moving some lookups that were being done in a stored procedure to perl. To do this I loaded the lookup tables to memory by creating hashes with the primary key as the hash key, and then emulating the code that created new rows in the lookup tables when there was a lookup miss. Among other things we wanted to keep track of what browsers people are using. There's a browser table which has a browser name, and an id. I loaded the table with:

```
$browsers = HashOfScalar($dbh, "select browser_id, browser
   from browsers",
                 'browser', 'browser_id');
```

And then the lookups like this:

```
$browser_id = getBrowser($browser);
```

Loading approximately 20 lookup tables to memory took about 30 megabytes of memory, but it decreased the run time of the log file parser from approximately 18 hours to about two hours, a quite favorable memory vs. speed tradeoff.

So what does HashOfScalar look like?

```
sub HashOfScalar {
        my ($dbh, $sql, $key, $val) = @_;

        my ($data, $restype, %row);

        $dbh->ct_execute($sql) == CS_SUCCEED or return undef;
        while($dbh->ct_results($restype) == CS_SUCCEED) {
                next unless $dbh->ct_fetchable($restype);
                # in this case we want to make sure that only
                # "normal" rows are placed in
                # the result array. Result rows from stored
                # procedure parameters, or
        # return status should be ignored.
        if($restype == CS_ROW_RESULT) {
            while(%row = $dbh->ct_fetch(CS_TRUE)) {
            if(!defined($row{$key})) {
            # Having a NULL key value is a problem for this
            # subroutine, as we would get potential collisions
            warn("Got a NULL value for $key in $sql - this is not
            supported");
            next;
            }
            # store the value in the $val column in the hash
            # at the index position represented by the $key column
            $data->{$row{$key}} = $row{$val};
            }
            } else {
            # ignore other types of results
            while(%row = $dbh->ct_fetch(1)) {
            ;
            }
        }
    }

    # return the hash reference that we've created here
    $data;
    }
```

And the getBrowser subroutine can be written this way:

```
    sub getBrowser {
        my $browser = shift;

        # return the existing browser_id if we have it:
        return $browsers->{$browser} if $browsers->{$browser};

        # this is a new browser, so create it in the database (assumes
        # $dbh is a global variable):
        my $browser_id = Scalar($dbh, "exec addBrowser '$browser'");
        $browsers->{$browser} = $browser_id;

        $browser_id;
    }
```

Of course, you can create all sorts of complex data types, for example, Arrays of Hashes, or Hashes of Hashes of Hashes (you could use the latter when the lookup key you wish to use is a two-level or two-column key.)

Writing a SybTools Module

Now this is all very fine, but still requires everyone using the code to understand the underlying Client Library structure. What we really want to do is create a module that all your programs (and programmers) can use. So let's create a perl module called SybTools, in which we will place these simplified access calls, but which will also allow us to use the standard Client Library calls.

We're going to get there in stages. First we create a SybTools.pm file, like this:

```
    # SybTools.pm

    package SybTools;
    use strict;
    use Sybase::CTlib;

    sub Scalar {
            # code from the Scalar sub described above...
    }

    sub HashRow {
            ...
    }

    1;

    __END__
```

Now, assuming this file is somewhere in perl's include path, you can write:

```
    Use SybTools;

    $dbh = new Sybase::CTlib $user, $pwd, $server;
    $date = SybTools::Scalar($dbh, "select max(date) from the_table");
```

Although this works, it's maybe a little more verbose than you really want it to be. We can improve on this by adding a few lines to the SybTools.pm file:

```
    # SybTools.pm

    package SybTools;
    use strict;
    use Sybase::CTlib;
    use Exporter;

    @SybTools::ISA = qw(Exporter);

    @SybTools::EXPORT = qw(Scalar HashRow);

    etc...
```

By adding the export lines we have now made the Scalar() and HashRow() subroutines visible in the namespace where “use SybTools” is called. So now you can write:

```
        Use SybTools;
...
        $date = Scalar($dbh, “select ... “);
```

In most cases this is good enough. But there’s one more step we can take. We can turn SybTools from a simple package module into a class of its own. By making SybTools a class, we make it extensible, and we make the syntax of calling SybTools subroutines identical to calling Sybase::CTlib subroutines.

Before we do so, however, here’s a quick review of method versus subroutine calls in perl. A method is a subroutine that is part of a “class”, as opposed to a subroutine that is merely part of a specific package. A method’s first parameter is either the package name, or the variable that is used to call it. For example, when you call

```
        $dbh = new Sybase::CTlib $user, $pwd, $server;
```

the first parameter that the new() subroutine gets is the string “Sybase::CTlib”. You can write the above line

```
        $dbh = Sybase::CTlib->new($user, $pwd, $server);
```

and get the exact same result. Similarily, when you call:

```
        $dbh->ct_execute(“select ...“);
```

the first parameter that ct_execute() receives is $dbh, which is often called $self in perl modules (but you can call it anything you want.)

To turn SybTools.pm into a class we need to make a couple of small changes: first we need to add “Sybase::CTlib” to the @ISA array, and we need to change the @EXPORT array. The @ISA array (for “is a”) tells perl which classes this class is derived from. By adding “Sybase::CTlib” to the @ISA array, we tell perl to look for methods in the Sybase::CTlib class if they are not found in the SybTools class. Then we’re going to set the @EXPORT array to be identical to the @EXPORT array of the Sybase::CTlib class, so that symbols such as CS_SUCCEED, etc., are visible to a program that uses SybTools.

So the first few lines of SybTools.pm now look like this:

```
# SybTools.pm
package SybTools;
use strict;
use Sybase::CTlib;
use Exporter;

@SybTools::ISA = qw(Sybase::CTlib Exporter)
@SybTools::EXPORT = @Sybase::CTlib::EXPORT;

sub Scalar {
        ...
}

etc...
```

Now we can write perl programs that access Sybase like this:

```
#!/usr/local/bin/perl -w
use strict;
use SybTools;
my $dbh = new SybTools $user, $pwd, $server;
my $date = $dbh->Scalar(“select max(date) from the_table“);
$dbh->ct_execute(“update the_table set date = getdate()“);

etc...
```

So what have we gained? First, you can see that we can avoid referencing Sybase::CTlib all together. All access to the Sybase::CTlib package is done via the SybTools package. This means that you can now replace the references to Sybase::CTlib in your scripts with SybTools, and you will automatically have access to the specialized access routines that you’ve written and placed in the SybTools.pm file. The syntax to access those routines is the same as the normal Sybase::CTlib syntax, so you and your users will not have to learn anything new.

In addition, you can easily add new routines to SybTools.pm and have all your users make use of them immediately. As you gain more confidence, it will also be possible for you to overload existing Sybase::CTlib routines with routines of your own, and thus transparently add functionality. You may, for example, wish to log all SQL traffic from your programs. This can be easily be done by writing a specialized ct_execute() subroutine in the SybTools module which stores the SQL that is being called to a file, and then calls the original ct_execute() subroutine. This change would be fully transparent to any program that uses SybTools instead of Sybase::CTlib directly.

The Next Step: Writing an Iterator

Quite often it is necessary to write a query that returns a large number of rows, and to process these rows one at a time. The normal way would be to call ct_execute() with the query, and then loop over ct_results() and ct_fetch() to get each row. Proper handling of ct_results() return values is sometimes a bit tricky, so maybe we can find a better way to do this.

We could create an iterator object that executed the query for us, and then returned one row at a time, each time we called the next() method, for example. It would be used like this:

```
$iter = $dbh->HashIter("select * from a_big_table");
while($row = $iter->next()) {
        # do something with the $row...
}
```

The HashIter() subroutine is a method in the SybTools class, but it is a special method in that it returns an object itself, rather than just returning data or a status value. Here is the code for HashIter(), which should be placed in the SybTools.pm file:

```
sub HashIter {
   my ($dbh, $sql) = @_;
   my $iter = {Handle => $dbh};
        # $iter is a reference to a hash, where one element
        # is the database handle
   $dbh->ct_execute($sql);
   my $restype;
   while($dbh->ct_results($restype) == CS_SUCCEED) {
        # if we've got fetchable data we break out of the loop
        last if $dbh->ct_fetchable($restype);
        }
   $iter->{LastResType} = $restype;
        # remember what the last ct_results() $restype was.
        # "bless" the $iter variable into the SybTools::HashIter package
        bless($iter, "SybTools::HashIter");
}

package SybTools::HashIter;
        # create a new namespace
sub next {
             my $self = shift;

             my %data;
             my $restype;

             loop: {
     %data = $self->{Handle}->ct_fetch(1);
     if(%data == 0) {
             # no more data in this result set - let's see if there is
             # another result set that has data....
        while($self->{Handle}->ct_results($restype) =CS_SUCCEED) {
             if ($self->{Handle}->ct_fetchable($restype)) {
                  # yes - we have fetchable data
                  $self->{LastResType} = $restype;
                  redo loop;
                          # jump to the 'loop:' tag above
                  }
             }
        return undef;
             # no more data - ct_results() returned something other
             # than CS_SUCCEED.
     }
             }

             { %data };    # return a reference to the data that we
                           # retrieved
     }
```

The code looks fairly complicated, but it is for a good cause. We've moved the complexity of handling multiple result sets, and of calling ct_results() the correct number of times out from the user's programs and into a library call, where it can be more thoroughly debugged. The user now only needs to call the HashIter() method, and then the next() method to get each row. The advantage when you have people writing perl code that are not familiar with Client Library should be fairly significant.

One additional benefit of creating a HashIter object is automatic cleanup. Specifically, if we add a DESTROY() subroutine to the SybTools::HashIter package, this subroutine will be automatically called when the HashIter object goes out of scope, or is undefined. This is important, as it is usually necessary to clean up a connection (by either calling ct_cancel() or by making sure that ct_results() has been called the appropriate number of times). This DESTROY() subroutine would look like this:

```
sub DESTROY {
        my $self = shift;

        $self->{Handle}->ct_cancel(CS_CANCEL_ALL);
}
```

Now you can do something like this (though this particular example is pretty silly):

```
sub getRecord {
        my $dbh = shift;
        my $iter = $dbh->HashIter(“select * from the_table“);
        my $row;

        while($row = $iter->next()) {
                last if($row->{User} eq ‘mpeppler‘);
        }

        $row;
}
```

When $iter goes out of scope in the getRecord() subroutine, the DESTROY() subroutine is automatically called, and the remainder of the query is automatically canceled.

Conclusion

The full source code for the SybTools module is available from www.isug.com or at at mpeppler@mbay.net. There is a sybperl mailing list: subscribe by sending a message to list-proc@list.cren.net with no subject and a body of: “subscribe Sybperl-L Your Name” where Your Name is your first and last name, of course. More information on perl is available from http://www.perl.com and http://www.perl.org. There is a quarterly perl journal called, appropriately enough, *The Perl Journal* (www.tpj.com).

There are quite a few perl books out there now. I’m partial to the O’Reilly offerings (*Programming Perl*—also known as the Camel book—and *Learning Perl*—the Llama) although I’ve heard good things about *Effective Perl Programming* (Hall, Addison-Wesley) too. ■

The Perl DBI (Data Base Interface) Module

A number of writers of database extensions to perl (Sybase, Oracle, Informix, Ingres, and others) started talking in early 1994 about writing a common API at the perl level for all databases. The API design work was done using e-mail and coordinated by Tim Bunce, who went on and wrote the DBI (Data Base Interface) module. This module creates a generic database access API, and allows the dynamic loading of drivers for various database engines. Drivers for different engines can coexist in a single perl program, which makes writing migration or data coordination programs where data is stored in various database engines quite easy. The API is quite simple, consisting mainly of a prepare/execute/fetch loop.
A typical DBI program might look like this:

```
#!/usr/local/bin/perl -w
use strict;
my $dbh = DBI->connect(“dbi:$driver:$additional_params“, $user,
$password);
my $sth = $dbh->prepare(“select * from the_table“);
$sth->execute;
my $data;
while($data = $sth->fetch) {
        print “@$data\n“;
}
$dbh->disconnect;
exit(0);
```

Assuming correct values for $driver, $additional_params, $user, and $password, this code should work unchanged for any DBI driver for which the SQL select statement is valid. There are limitations: If the SQL that is used in a DBI program is non-standard, then the program becomes non-portable, but at least the perl side of the program is portable.

The Sybase driver for DBI (DBD::Sybase) is currently in alpha release, and implements most of the DBI spec, with some extensions to allow for the retrieval of multiple result sets for a single execute operation, as this is something that happens quite often when you execute various Transact-SQL constructs. The DBD::Sybase module is not distributed as part of the sybperl package, but is available from the same place where you got sybperl. DBD::Sybase is written using Client Library, so it will not work with systems that do not have at least OpenClient 10.x installed.

Should DBD::Sybase be preferred over sybperl?

I personally think not. I prefer to use the native API, as this gives me better control over what I do. But if database portability is an issue for you, or if you need to access different database engines from your programs, then I would say that DBD::Sybase, although still labeled as alpha code, is certainly a good way to go. ■