Cut-to-the-Chase Series - **Perl Programming**

Misc Topics

Perl Programming – Misc Topics

Table of Contents

Misc Topics

"Knowledge is of two kinds. We know a subject ourselves, or we know where we can find information upon it."

-Samuel Johnson

ICON KEY
Toy Code
Hands-on
Important Info
Be Careful

This Module

It is assumed that you have already finished all of the foundation topics, or have a very solid understanding of Perl. It is also highly recommended that you have finished the module on 'References'.

About this Module

This is collection of topics that either did not fit in well anywhere else, or whose materials were too short to warrant their own module at this time. I feel compelled to cover these subjects as they will serve to round out your knowledge of the language.

This modules is not intended to provide comprehensive coverage to these subjects. The goal is to provide initial exposure and turn you on to some additional resources.

Eval( )

Eval is a very interesting animal that warrants its own heading. Eval is a function that treats an expression like a Perl program and executes it. At face value this may not seems that that big a deal. First, lets take a look at a very simple program.

EVAL1.pl

```
my $str = "hello world";

my $cmd = 'print $str;';

eval $cmd;
```

Up until now we have been a bit loose with the use of single or double quotes. In many cases it does not matter which type you use. With single quoted strings variable interpolation does not occur. I have set a scalar to the string 'print $str;'. This string represents a valid statement in Perl. I can pass this statement to the eval function and it will execute and return to STDOUT (the default) "hello world".

This raises some interesting possibilities and uses for us. Eval could be used as a code validator. We can pass a Perl program or program fragment or regular expression to Eval to validate it.

By using Eval we can accept code dynamically. With Eval it becomes possible to writing programs that are adaptive (can be reprogrammed in mid flight). We can write programs that accept code stubs at runtime. As an example, we can write programs that accept and evaluate "on the fly" regular expressions. The following is a some toy code that demonstates this point.

EVAL2.pl

```
use strict;
my $prog = '
            my %hsh =
            (
            firstname => "John",
            middle_init => "H.",
            lastname => "Smith",
            );

            my $key="";
            my $val="";
            while (($key, $val) = each(%hsh))
            {
              print "$key => $val\n";
            }
           ';

if (eval($prog)) { } else  { print $@; }
```

There are a number of things to point out and discuss with this program. First, notice that we have an entire script within a scalar.

```
my $prog = '
            my %hsh =
            (
            firstname => "John",
            middle_init => "H.",
            lastname => "Smith",
            );
            my $key="";
            my $val="";
            while (($key, $val) = each(%hsh))
            {
              print "$key => $val\n";
            }
             ';
```

Do not forget to use the single quotes! Also keep in mind not to use single quotes within the program you have contained in the scalar. Since I am running a program from within another program I need some means of trapping syntax error. The following code fragment does this.

```
if (eval($prog)) { } else  { print $@; }
```

I will agree that this logic looks a little obscure, though it gets the job done. The eval function will be true if it can execute our program (or snippet). If not, we execute the else which will return the contents of $@ which is the returned error. We do not need any code within the "if" premise as eval with return to STDOUT as a standard default.

Working with Excel

I really should be calling this "creating and using com servers from Perl", but that title might not get your attention, and I am not going to go into the specifics regarding that subject. Our focus here will be to demonstrate how to work with Excel spreadsheets from a Perl program using the Win32::OLE module.

There exist some great tools and programs on the 'Wintel' platform. Perl is a great glue language and it has very nice and clean interoperability with Windows programs. Since my bias for computing is this platform I would be remiss if I did not show you these. For most of you, even if you work on mainframes or servers running linux or Unix, it is a safe bet that many of you do your computing on the PC and use tools like Microsoft Office.

Where I work many of us pound away on a terminal that accesses our Compaq NonStop development system and it's operating system named Guardian90. Of course we do this through a PC that is connected to our corporate network. We use Office and Lotus Notes as our email system. We use Perl on Guardian, on various Linux servers, and on NT and Windows 9x systems. Many folks use Access and Excel on the PC to manage their work and the efforts of the various teams. Perl's ability to access this data makes it a real plus. With Perl it is possible for me to get data from our NonStop system and dump it into an Excel spreadsheet. The possibilities for use with this language are endless.

Creating a com server from a Perl program is a remarkably simple venture. The following is a program that reads a group of cells from a spreadsheet.

EXCEL1a.pl / BOOK1a.xls*

* Or you can tweak the code and use another spreadsheet

Take a look at the code, run the program, then I will break it down and discuss it with you.

```
use Win32::OLE;


my $com_obj = "Excel.Application";
# do not use convention "c:/myperl/book1.xls"
my $ComFile ="c:\\myperl\\book1a.xls";


my $ComServer = Win32::OLE->GetActiveObject( $com_obj );
```

```
if( ! $ComServer)
{
  $Excel = new Win32::OLE( $com_obj, \&QuitApp )
   or die "Could not create a COM '$com_obj' object";
}

# open excel file
my $workbook = $ComServer->Workbooks->Open($ComFile);
#open worksheet by numeric reference
my $worksheet = $workbook->Worksheets(1);

# retrieve values from worksheet
for (my $i=1; $i<=32; $i++)
{
  my $col = "B";
  $col .= $i;
  my $b = $worksheet->Range($col)->{'Value'};
  print "$b\n";
}

# stop com server
sub QuitApp
{
  my( $ComObject ) = @_;
  print "Quitting " . $ComObject->{Name} . "\n";
  $ComObject->Quit();
}
```

First things first, we need to reference the OLE module. As a historical note, there is an OLE module that you can still use. If you look at the code it uses the Win32::OLE module. It is kept around for backwards compatibility with older Perl code. You may run into examples that reference this module. There are differences in syntax between the two modules. You should use Win32::OLE.

```
use Win32::OLE;
```

Next, I am using a scalar and assigning it to the type of com object I plan on instancing.

```
my $com_obj = "Excel.Application";
```

To access Word you would say… "Word.Application"; To access Powerpoint you would say "Powerpoint.Application"…you get the point. We could hardcode this value within other areas of our program, though I have opted to use a scalar. Next, I create a scalar and assign it to the file I want.

```
my $ComFile ="c:\\myperl\\book1a.xls";
```

From here we need to write the code that actually will create the com server.

```
my $ComServer = Win32::OLE->GetActiveObject($com_obj);

if( ! $ComServer) {

  $ComServer = new Win32::OLE($com_obj, \&QuitApp)

   or die "Could not create a COM '$com_obj' object";

}
```

A number of things are happening in the preceding code. Our conditional is looking to see if Excel is already running. If it is that instance is what we will use. Hey, why incur the overhead and resources of starting the program again. If not we will start Excel. You will not notice that is has been started. The default is “not visible”. If you want to actually see the program start you could add the line:

```
$ComServer->{'Visible'} = 1;
```

If for some reason we cannot start this com server (you don’t have Excel on your machine!) we will die with an error message. Now, quickly back to the line of code:

```
  $ComServer = new Win32::OLE($com_obj, \&QuitApp)
```

We are creating new com server. We pass in two arguments. The first is the type of server we want. The second is a reference to our subroutine. The second argument is used by the modules

Quit( ) method. We must pass a reference so the module can bless it. Lets skip to this subroutine.

```
sub QuitApp {

  my($ComObject) = @_;

  print "Quitting " . $ComObject->{Name} . "\n";

  $ComObject->Quit();

}
```

QuitApp( ) ensures that we properly clean up. @_ contains a reference to a hash containing all the parameter, attributes, et al related to our com server. It is important that $ComObject be enclosed in parenthesis. The QuitApp( ) sub and the code that creates the com server is pretty boiler-plate stuff so it is not all that important that you understand explicitly how it works. So, if some of the explanation is not clicking, do not worry about it.

Since Win32::OLE is an objected oriented module, you may have noticed that we are using references to the various method calls.

We can now look at the code that manipulates the spreadsheet.

Access a Workbook

```
my $workbook = $ComServer->Workbooks->Open($ComFile);
```

This is how we access a workbook.

Reference a Worksheet

```
my $worksheet = $workbook->Worksheets(1);
```

This is how we access a worksheet. 1 represents the first worksheet, 2 would be the second one, and so on.

Retrieving values from cells

```
for (my $i=1; $i<=32; $i++) {

  my $col = "B";
  $col .= $i;

  my $b = $worksheet->Range($col)->{'Value'};

  print "$b\n";

}
```

There are a number of means of retrieving values from cells. I have chosen this method to show you first as the other method involves dereferencing and anonymous array of anonymous arrays (a bit dicey). In this example I am passing the column I want into "Range", retrieving the value and then printing it.

General information pertaining to syntax

Com follows a hierarchical object model. Up to this point I have not spend a lot of time analyzing and discussing the syntax. I have just told you that if you want to retrieve the value of a cell (or range of cells) you code…

my $b = **$worksheet->Range(*<cells you want>*)->{'Value'};**

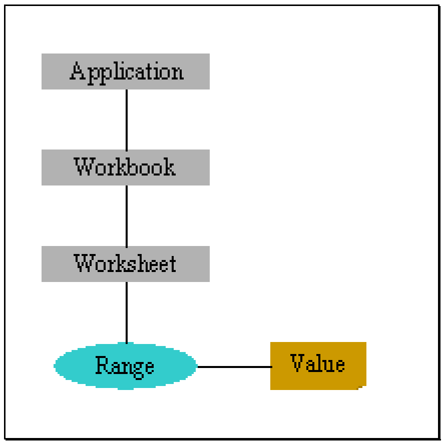
$worksheet is a reference to our worksheet.

my $worksheet = **$workbook->Worksheets(1)**;

And $workbook is a reference to our spreadsheet.

my $workbook = **$ComServer->Workbooks->Open($ComFile);**

To look at the object hierarchy another way…



You can get the object hierarchies from www.microsoft.com. You will need to translate their syntax into Perl syntax. Here is an example.

Microsoft speak:

Range("B1").Value

Perl speak:

Range("B1")->{'Value'}

In general you replace the "." with "->". It will take some time an experience to get used to this but it is well worth it. While Perl is second to none in terms of text processing, being able to access Windows applications and databases from Perl opens up a whole new horizon for you!

More on retrieving data

In the last example I chose to write a loop and to retrieve data one cell at a time. This method can serve to give you more iterative control it may not be the best method to use. I am assuming you are familiar with Excel and understand that you can reference a group of cell by writing something like, "A1:C3". This expression will select a range of cell as depicted in the following table.

	A	B	C	D
1	🟦	🟦	🟦	
2	🟦	🟦	🟦	
3	🟦	🟦	🟦	
4				

You can use the "Range" method to select a group of cells.

EXCEL1b.pl

```
  my $b = $worksheet->Range("B1:B32")->{'Value'};
```

In this case $b is a reference to an anonymous array. It just so happens that the anonymous array contains references to other anonymous arrays which hold the data for each cell. In order to dereference and get at the data we can write a nested foreach loop.

```
  foreach my $CellsArr (@$b)   {

    foreach my $CellVals (@$CellsArr)   {

       print "$CellVals\n";

    }

  }
```

You can see why I strongly recommended that you completed the module on “References”. This stuff can get quite interesting.

Updating an existing spreadsheet

Spreadsheets are a nice means for collecting and storing data that is of a ‘flat-file’ nature. Spreadsheets also are very strong at formulaic operations. It is conceivable you might want to create a spreadsheet just to manipulate data with a formula that is easier to create in Excel. You could also use a spreadsheet to process ‘what if’ scenarios on data. First, is a simple example, followed by two examples that will hopefully get your imagination in full trot.

For the examples that follow you may find it more convenient to have the spreadsheet Book1a.xls up and running so you can watch what happens when you run the Perl script.

A trivial example

🖫 EXCEL2a.pl

Writing to a cell or group of cells is a pretty straight-forward operation.

```
my $Range = $worksheet->Range("E9");

$Range->{Value} = "Throw momma from the train";
```

I have created two generic subroutines to write to the file. One which simulates the ‘Save’ command, the other for the ‘Save As’ command.

Save

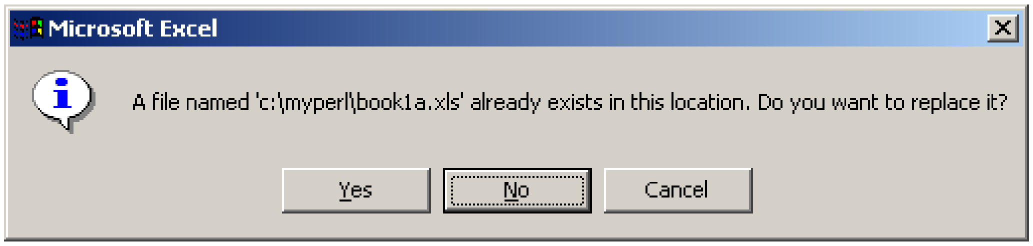
```
sub Save

{
  $workbook->Save();

}
```

'Save' wants no argument and accepts no arguments

Save As

```
sub SaveAs

{

  $workbook->SaveAs($ComFile);

}
```

'SaveAs' accepts the name of the exisiting file or the name of a new file you want to create. Be advised that the dialog box you get when rewriting the existing file is a bit misleading.



Microsoft Excel

A file named 'c:\myperl\book1a.xls' already exists in this location. Do you want to replace it?

You are not replacing the file, you are updating it!

A more visual Array / User defined parameters

This is a nice little example that demonstrates how you could use a spreadsheet to gather parameters that are user defined. Face the music, a spreadsheet does provide a nice visual interface and Perl, while enormously useful, is 'command-line' ugly!

EXCEL2b.pl

In this example I have a small table. The first row represents the numbers I want to calculate. The second row represents the calculation factor. The last row is the product of the first two rows [formula is: =SUM(D1*D2) etc…]

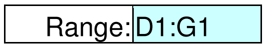
27	22	12	11
1	1.25	1.5	1.75
27	27.5	18	19.25

I pass in the first row of numbers from my Perl program. Here is the code.

```
#$Range->{Value} = 11.576;

#$Range->{Value} = [22,12]; #see what happens?

$Range->{Value} = [10, 6, 17, 2]; #list of values

Save();
```

I have a few variations commented out that you can uncomment and play with.

The next code fragment allows the user to determine which cells in the spreadsheet will get updated.



Range:	D1:G1

```
my $CellRange = $worksheet->Range("E5")->{'Value'};

my $Range = $worksheet->Range($CellRange);
```

Great stuff!

Some more formatting

As I began to document this I realized that there is no end to the volume of stuff that can be documented. I am going to provide another example related to formatting then it is up to you to do any further digging. “VBAXL9.chm” is the VBA reference for Excel. “XLMAIN9.chm” is the Excel reference.

EXCEL2c.pl / BOOK1b.xls

```
$worksheet->Columns("A")->{AutoFit};  #or...
#$worksheet->Columns("A")->{ColumnWidth} = 60;


$worksheet->Cells(1,1)->{Value} = "Welcome to some more formatting";
$worksheet->Cells(1,1)->Font->{Size} = 16;
$worksheet->Cells(1,1)->Font->{ColorIndex} = 22;
```

```
$worksheet->Cells(2,1)->{Value} = "Hello";
$worksheet->Cells(2,1)->Font->{Bold} = "True";
$worksheet->Cells(2,1)->Font->{Name} = "Bookman Old Style";

$worksheet->Cells(3,1)->{Value} = "World";
$worksheet->Cells(3,1)->Font->{Italic} = "True";

$worksheet->Cells(4,1)->{Value} =  75;
$worksheet->Cells(4,1)->{HorizontalAlignment} = "xlHAlignJustify";

$worksheet->Cells(5,1)->{Value} =  3000433.87877887;
$worksheet->Cells(5,1)->{NumberFormat} = "$###,##.##";
```

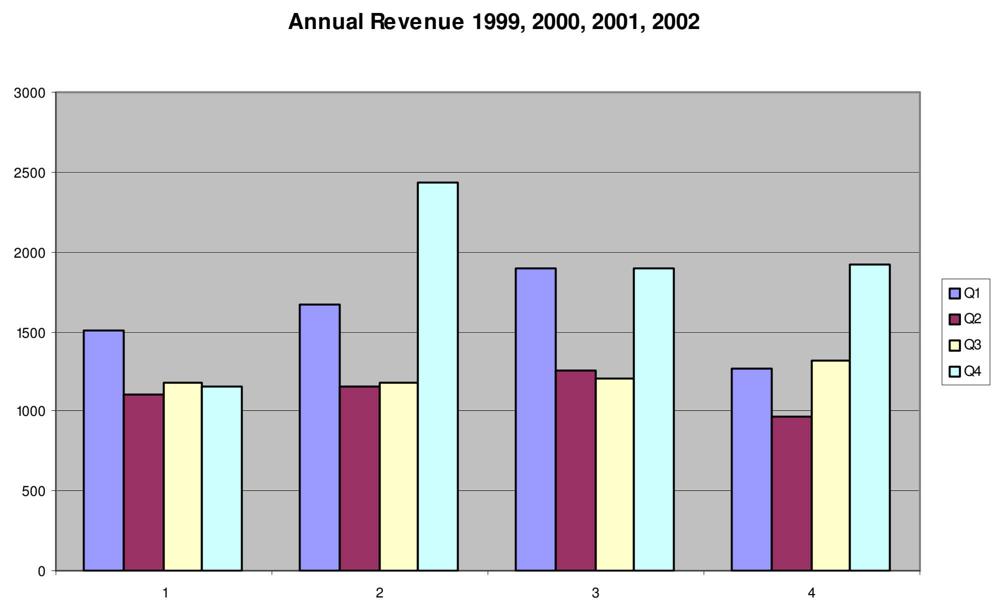
Creating a chart/graph

I put this together at the request of a friend and co-worker that asked that this be a part of a class I am putting together. This is pretty bare-bones, but will give you a taste of what you can do

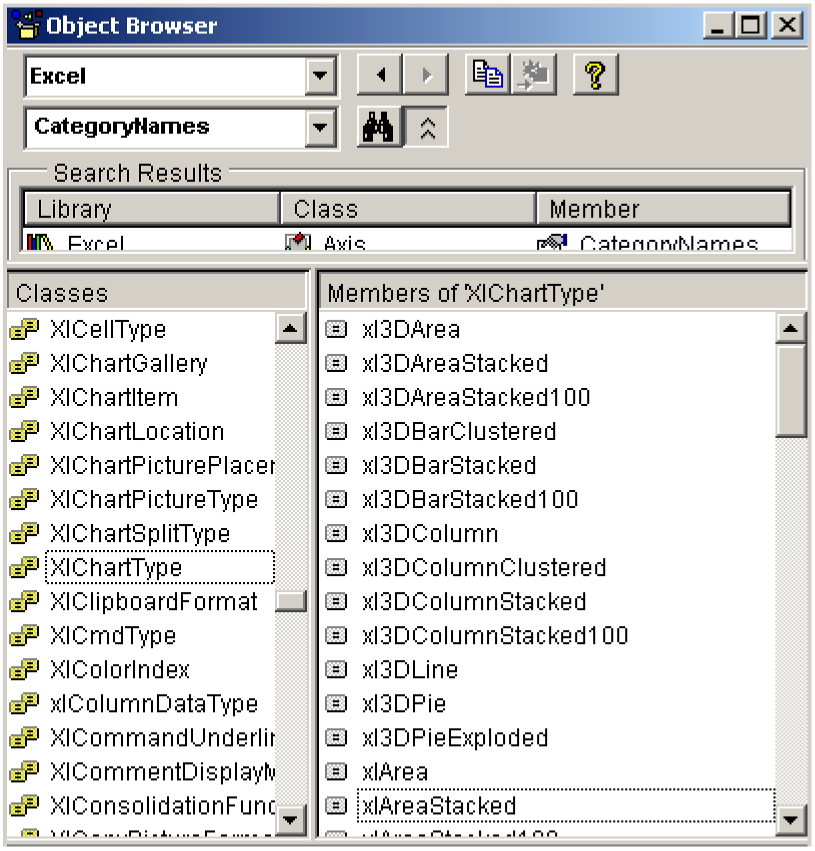
EXCEL3.pl

The following code will produce the following chart.

```
my $Range = $worksheet->Range($CellRange);
$Range->{Value} =
    [['Q1', 'Q2', 'Q3', 'Q4'],
     [1504, 1102, 1186, 1154],
     [1670, 1150, 1174, 2433],
     [1891, 1261, 1201, 1899],
     [1274, 971,  1321, 1922], ];
Save();
CreateChart();
sub CreateChart {
 my $Chart = $ComServer->Charts->Add;

 $Chart->{ChartType} = 'xlAreaStacked';

 $Chart->SetSourceData({Source => $Range, PlotBy => 'xlColumns'});

 $Chart->{HasTitle} = 1;

 $Chart->ChartTitle->{Text} = "Annual Revenue 1999, 2000, 2001,
2002";

}
```



The object browser that comes with Office can be used to see what methods and properties are available.



Library	Class	Member
Excel	Axis	CategoryNames

XlCellType
XlChartGallery
XlChartItem
XlChartLocation
XlChartPicturePlacer
XlChartPictureType
XlChartSplitType
XlChartType
XlClipboardFormat
XlCmdType
XlColorIndex
xlColumnDataType
XlCommandUnderlir
XlCommentDisplayM
XlConsolidationFunc

xl3DArea
xl3DAreaStacked
xl3DAreaStacked100
xl3DBarClustered
xl3DBarStacked
xl3DBarStacked100
xl3DColumn
xl3DColumnClustered
xl3DColumnStacked
xl3DColumnStacked100
xl3DLine
xl3DPie
xl3DPieExploded
xlArea
xlAreaStacked

Creating a brand new spreadsheet

There is a very handy nice module named "Spreadsheet::WriteExcel". This module allows you to create Excel spreadsheets from scratch. Another nice advantage of this module is that it works on other operating systems (Linux, Unix, etc…) and does not require that you have Excel. Of course you will need Excel or some other program that can read Excel files in order to view the results.

This modules is very well documented (61 pages of good stuff). It comes with a wealth of code examples and web documentation. Since this is the case I see no reason to reinvent the wheel and refer you there.

ODBC

When you couple the ability to create ole servers and ODBC and network communications (telnet, ftp, etc…) with the fantastic text processing capabilities of Perl you end up with a truly remarkable environment for solving problems.

At first I must admit that I did not think much of using Perl for access to structured data. After all, that is what SQL is for. But, you have the ability to write and execute SQL queries within your Perl program. This opens up a whole new dimension to the language.

I am going to show you a few examples. It is assumed that you already know how to set up ODBC sources. You also will need to install the Win32::ODBC module. All of my examples are based on using the Northwind database and Microsoft Access. I choose this because of the widespread popularity of this database. I used the Northwind database as it comes with the software so everyone that has it will have this database. You may need to search and find it on your system.

You can use Win32::ODBC with any ODBC data source that you have a driver for. Having said that here is one big disclaimer. Databases all have slightly to radically different implementations. Just because it uses the buzzword SQL does not mean that is sports the same functionality as the next database. This is a sad but true fact. You may try functionality with this module that works with one database but not with another. You have been forewarned.

Using Win32::ODBC is a fairly straight forward affair. First, I will give you the basics, then I will walk you through a few examples.

The Basics

These are designed to give you the basic syntax for connecting to databases in Perl via ODBC.

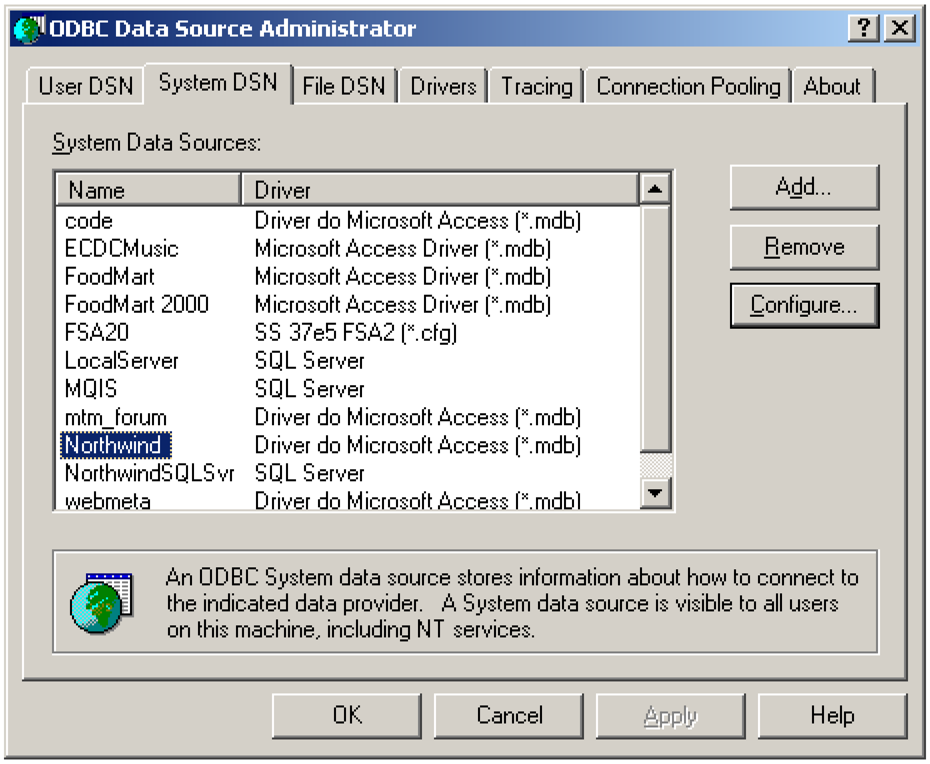
Creating an ODBC object

Syntax

```
my $db = new ODBC("<name_of_odbc_database>");
```

Example

```
my $db = new ODBC("Northwind");
```



Name	Driver
code	Driver do Microsoft Access (*.mdb)
ECDCMusic	Microsoft Access Driver (*.mdb)
FoodMart	Microsoft Access Driver (*.mdb)
FoodMart 2000	Microsoft Access Driver (*.mdb)
FSA20	SS 37e5 FSA2 (*.cfg)
LocalServer	SQL Server
MQIS	SQL Server
mtm_forum	Driver do Microsoft Access (*.mdb)
Northwind	Driver do Microsoft Access (*.mdb)
NorthwindSQLSvr	SQL Server
webmeta	Driver do Microsoft Access (*.mdb)

An ODBC System data source stores information about how to connect to the indicated data provider. A System data source is visible to all users on this machine, including NT services.

Writing an SQL query in Perl

Syntax

```
my $sql = “<query_code_goes_here>”;
```

Example

```
my $sql="
              SELECT *
              FROM Customers
              WHERE ContactTitle = 'Owner'
        ";
```

Sql method

Syntax

```
if ($db->Sql(<sql_query_goes_here>))
```

Example

```
if ($db->Sql($sql))
```

FetchRow / DataHash methods

Syntax

```
$<obbc_object_reference>->FetchRow()

%<hash> = $db->DataHash();
```

Please make note that FetchRow() is just as the name implies, "one row at a time".

Example

```
    while ($db->FetchRow() )

   {

     # dump records into a hash

     %customer = $db->DataHash();

     print_row();

   }
```

Close method

Syntax / Example

```
$db->Close();
```

Examples

Here are a few examples.

Accessing Access Objects for a Database

🖫 ODBC_CATALOG.pl

This one produces a list of database objects using the Catalog( ) method.

```
use strict;
use Win32::ODBC;

#refer to DSN, must have database defined through ODBC
my $db = new ODBC("Northwind");
my $iCnt=0;

# Catalog(<database>,<owner>,<table>,<object type>)
# object type parameters:
# TABLE
# VIEW
# SYSTEM TABLE
# GLOBAL TEMPORARY
# LOCAL TEMPORARY
# ALIAS
# SYNONYM
# Note: these work dependent on database vendor
my $objtyp = " 'SYSTEM TABLE', 'TABLE' ";

if ( $db->Catalog("","","%","$objtyp") )
{
  while ($db->FetchRow() )
  {
    my %metadata = $db->DataHash();
    print "$metadata{TABLE_NAME} $metadata{TABLE_TYPE} \n";
  }
}
```

```
$db->Close();
```

Simple SQL Query

🖫 ODBCSELECT1a.pl

This is a simple select that returns a filtered recordset.

```
use strict;
use Win32::ODBC;
my $iRow=0;
my $sql="
              SELECT *
              FROM Customers
              WHERE ContactTitle = 'Owner'
        ";

#refer to DSN, must have database defined through ODBC
my $db = new ODBC("Northwind");

# execute an SQL statement
if ($db->Sql($sql))
{
  print "SQL select error";
}
else
{
# process resultant recordset
   while ($db->FetchRow() )
   {
     # dump records into a hash
     my %customer = $db->DataHash();
     $iRow++;
                           # reference column
     print "$iRow COMPANY: $customer{CompanyName}"
```

```
              . " "
              . "CONTACT: $customer{ContactName} \n";
   }
}


$db->Close();
```

Looping through a recordset

🖫 ODBCSELECT1b.pl

This is a simple select that returns all rows.

```
use strict;
use Win32::ODBC;
my $sql="
              SELECT *
              FROM Customers
              WHERE ContactTitle = 'Owner'
        ";
my %customer;

#refer to DSN, must have customerbase defined through ODBC
my $db = new ODBC("Northwind");

# execute an SQL statement
if ($db->Sql($sql))
{
  print "SQL select error";
}
else
{
# process resultant recordset
# The point behind this is to demonstrate through code that as function
call
# name implies, we are only fetching one row of customer at a time
```

```
    while ($db->FetchRow() )
   {
     # dump records into a hash
     %customer = $db->DataHash();
     print_row();
   }
}

sub print_row
{
 my $key;
 my $value;
 foreach $key (keys %customer)
 {
  ($key, $value) = (each %customer);
  if (!$value)
  {
    $value = " "; #to avoid unitialized value warning for null fields

  }
  print "$key: $value\n";
 }
}

$db->Close();
```

Getting a list of database objects

🖫 ODBC_OBJLIST.pl

This one returns a list of all the views that start with the name "Product".

```
use strict;
use Win32::ODBC;


#refer to DSN, must have database defined through ODBC
```

```
my $db = new ODBC("Northwind");

# TableList(<database>,<owner>,<table>,<object type>)
# object type parameters:
# TABLE
# VIEW
# SYSTEM TABLE
# GLOBAL TEMPORARY
# LOCAL TEMPORARY
# ALIAS
# SYNONYM
# Note: these work dependent on the database vendor you are dealing with

my $objtyp = " 'VIEW' ";
my @tbl_list = $db->TableList("","","PRODUCT%",$objtyp);

for (my $i=0;$i<@tbl_list; $i++)
{
  print $tbl_list[$i] . "\n";
}

$db->Close();
```

Getting OBDC Attributes

This is a pretty laborious one. I am not going to cover any of the code. I picked it up from the web somewhere and am including it here.

DBINFO.pl

```
>perl dbinfo.pl northwind
```

```
use Win32::ODBC;


$DSN = "Test" unless $DSN = $ARGV[0];
```

```
$db = new Win32::ODBC( $DSN )
    ||    die    "Error:    Could    not    connect    to    \"$DSN\".\n"    .
Win32::ODBC::Error() . "\n";


@Types = qw(
            BIGINT
            BINARY
            BIT
            CHAR
            DATE
            DECIMAL
            DOUBLE
            FLOAT
            INTEGER
            LONGVARBINARY
            LONGVARCHAR
            NUMERIC
            REAL
            SMALLINT
            TIME
            TIMESTAMP
            TINYINT
            VARBINARY
            VARCHAR
);


%Attributes = (
                "Position" => {
                    Desc => "Postioned operations",
                    Value   =>  'SQL_POS_OPERATIONS',
                    Function =>  GetInfo,
                    Position => 'SQL_POS_POSITION',
                    Refresh =>  'SQL_POS_REFRESH',
                    Update  =>  'SQL_POS_UPDATE',
                    Delete  =>  'SQL_POS_DELETE',
                    Add      =>  'SQL_POS_ADD'
                    },
                "Position Statements" => {
                    Desc     =>  "Positioned Statements",
                    Value    =>  'SQL_POSITIONED_STATEMENTS',
                    Function =>  GetInfo,
                    Delete   =>  'SQL_PS_POSITIONED_DELETE',
                    Update   =>  'SQL_PS_POSITIONED_UPDATE',
                    "Select Update" => 'SQL_PS_SELECT_FOR_UPDATE'
                    },
                "Row Updates"       =>  {
                    Desc     =>  "Can detect changes made by other users",
                    Value    =>  'SQL_ROW_UPDATES',
                    Function =>  GetInfo
                    },
                "Server Name"       =>  {
                    Desc     =>  "Name of database server",
                    Value    =>  'SQL_SERVER_NAME',
                    Function =>  GetInfo
                    },
                "Special Characters"=>  {
                    Desc       =>    "Non  letter  chars  that  can  be  used
legally",
                    Value    =>  'SQL_SPECIAL_CHARACTERS',
                    Function =>  GetInfo
                    },
                "Driver Name"   =>  {
                    Desc     =>  "ODBC Driver Name",
                    Value    =>  'SQL_DRIVER_NAME',
                    Function =>  GetInfo
                    },
                "Driver Version"    =>  {
                    Desc     =>  "ODBC Driver Version",
                    Value    =>  'SQL_DRIVER_VER',
                    Function =>  GetInfo
                    },
                "Driver ODBC Version"   =>  {
                    Desc    =>  "ODBC Version this driver supports",
                    Value   =>  'SQL_DRIVER_ODBC_VER',
                    Function =>  GetInfo
                    },
                "Order by expression"   =>  {
                    Desc     =>  "Can this driver expressions in the ORDER
BY list",
                    Value   =>  'SQL_EXPRESSIONS_IN_ORDERBY',
                    Function =>  GetInfo
                    },
                "Maximum active statements" =>  {
                    Desc        =>    "Max  number  of  active  statements
connection (0 if no limit)",
                    Value   =>  'SQL_ACTIVE_STATEMENTS',
                    Function =>  GetInfo
                    },
                "DSN"           =>  {
                    Desc    =>  "Data Source Name",
                    Value   =>  'SQL_DATA_SOURCE_NAME',
                    Function =>  GetInfo
                    },
                "DSN Read Only" =>  {
                    Desc    =>  "Is this DSN read only?",
                    Value   =>  'SQL_DATA_SOURCE_READ_ONLY',
                    Function =>  GetInfo
                    },
                "Database Name" =>  {
                    Desc    =>  "Name of this database",
                    Value   =>  'SQL_DATABASE_NAME',
                    Function =>  GetInfo
                    },
                "Cursor Rollback"   =>  {
                    Desc    =>  "Behavior of a Cursor Rollback",
                    Value   =>  'SQL_CURSOR_ROLLBACK_BEHAVIOR',
                    Function =>  GetInfo,
                    Delete  =>  'SQL_CB_DELETE',
                    Close   =>  'SQL_CB_CLOSE',
                    Preserve=>  'SQL_CB_PRESERVE'
                    },
                "Cursor Commit"     =>  {
                    Desc    =>  "Behavior of a Cursor Commit",
                    Value   =>  'SQL_CURSOR_COMMIT_BEHAVIOR',
                    Function =>  GetInfo,
                    Delete  =>  'SQL_CB_DELETE',
                    Close   =>  'SQL_CB_CLOSE',
                    Preserve=>  'SQL_CB_PRESERVE'
                    },
                "Fetch Direction"   =>  {
                    Desc    =>  "Directions that Fetch Supports",
                    Value   =>  'SQL_FETCH_DIRECTION',
                    Function =>  GetInfo,
                    Next    =>  'SQL_FD_FETCH_NEXT',
                    First   =>  'SQL_FD_FETCH_FIRST',
                    Last    =>  'SQL_FD_FETCH_LAST',
                    Prior   =>  'SQL_FD_FETCH_PRIOR',
                    Absolute=>  'SQL_FD_FETCH_ABSOLUTE',
                    Relative=>  'SQL_FD_FETCH_RELATIVE',
#                       Resume  =>  'SQL_FD_FETCH_RESUME',
                    Bookmark=>  'SQL_FD_FETCH_BOOKMARK'
                    },
                "File Usage"    =>  {
                    Desc    =>  "How driver treats files in a data source
(if driver is single tier)",
                    Value   =>  'SQL_FILE_USAGE',
                    Function =>  GetInfo,
                    "Not Supported" =>  'SQL_FILE_NOT_SUPPORTED',
                    Table   =>  'SQL_FILE_TABLE',
                    Qualifier   =>  'SQL_FILE_QUALIFIER'
                    },
                "Case sensitivity"      =>  {
                    Desc    =>  "How case affects SQL identifers",
                    Value   =>  'SQL_IDENTIFIER_CASE',
                    Function =>  GetInfo,
                    Upper   =>  'SQL_IC_UPPER',
                    Lower   =>  'SQL_IC_LOWER',
                    Sensitive   =>  'SQL_IC_SENSITIVE',
                    Mixed   =>  'SQL_IC_MIXED'
                    },
                "Quote Character"   =>  {
                    Desc      =>   "Character  string  used  as  starting  &
ending deliminator for quoting",
                    Value   =>  'SQL_IDENTIFIER_QUOTE_CHAR',
                    Function =>  GetInfo
                    },
                "SQL Reserved Keywords" =>  {
                    Desc    =>  "Keywords reserved for SQL use",
                    Value   =>  'SQL_KEYWORDS',
                    Function =>  GetInfo
                    },
                "Time/Date SQL functions"   =>  {
                    Desc       =>    "SQL  Time/Date  Functions  that  are
supported by this ODBC driver",
                    Value   =>  'SQL_TIMEDATE_FUNCTIONS',
                    Function =>  GetInfo,
                    "Current Date"  =>  'SQL_FN_TD_CURDATE',
                    "Current Time"  =>  'SQL_FN_TD_CURTIME',
                    "Day Name"  =>  'SQL_FN_TD_DAYNAME',
                    "Day of Month"  =>  'SQL_FN_TD_DAYOFMONTH',
                    "Day of Week"   =>  'SQL_FN_TD_DAYOFWEEK',
                    "Day of Year"   =>  'SQL_FN_TD_DAYOFYEAR',
                    Hour    =>  'SQL_FN_TD_HOUR',
                    Minute  =>  'SQL_FN_TD_MINUTE',
                    Month   =>  'SQL_FN_TD_MONTH',
                    "Month Name"    =>  'SQL_FN_TD_MONTHNAME',
                    Now     =>  'SQL_FN_TD_NOW',
                    Quarter =>  'SQL_FN_TD_QUARTER',
                    Second  =>  'SQL_FN_TD_SECOND',
                    "TimeStamp Add"         =>  'SQL_FN_TD_TIMESTAMPADD',
                    "TimeStamp Difference"  =>  'SQL_FN_TD_TIMESTAMPDIFF',
                    Week    =>  'SQL_FN_TD_WEEK',
                    Year    =>  'SQL_FN_TD_YEAR'
                    },
                "User Name" =>  {
                    Desc    =>  "Userid of current user",
                    Value   =>  'SQL_USER_NAME',
                    Function =>  GetInfo
                    },
                "SQL Union support" =>  {
                    Desc     =>  "Does this ODBC Driver support the UNION
clause?",
                    Value   =>  'SQL_UNION',
                    Function =>  GetInfo,
                    "Union Support" =>  'SQL_U_UNION',
                    "Union 'ALL' Keyword Support"   =>  'SQL_U_UNION_ALL'
                    },
                "SQL String Functions"  =>  {
                    Desc    =>  "SQL String Functions that are supported",
                    Value   =>  'SQL_STRING_FUNCTIONS',
                    Function =>  GetInfo,
                    Ascii =>  'SQL_FN_STR_ASCII',
                    Char =>  'SQL_FN_STR_CHAR' ,
                    Concat =>  'SQL_FN_STR_CONCAT',
                    Difference =>  'SQL_FN_STR_DIFFERENCE',
                    Insert =>  'SQL_FN_STR_INSERT',
                    LCase =>  'SQL_FN_STR_LCASE',
                    Left =>  'SQL_FN_STR_LEFT',
                    Length =>  'SQL_FN_STR_LENGTH',
                    Locate =>  'SQL_FN_STR_LOCATE',
                    Locate_2 =>  'SQL_FN_STR_LOCATE_2',
                    LTrim =>  'SQL_FN_STR_LTRIM',
                    Repeat =>  'SQL_FN_STR_REPEAT',
                    Replace =>  'SQL_FN_STR_REPLACE',
                    Right =>  'SQL_FN_STR_RIGHT',
                    RTrim =>  'SQL_FN_STR_RTRIM',
                    Soundex =>  'SQL_FN_STR_SOUNDEX',
                    Space =>  'SQL_FN_STR_SPACE',
                    Substring =>  'SQL_FN_STR_SUBSTRING',
                    UCase =>  'SQL_FN_STR_UCASE'
                    },
                "SQL Numeric Functions" =>  {
                    Desc        =>    "SQL  Numeric  Functions  that  are
supported",
                    Value   =>  'SQL_NUMERIC_FUNCTIONS',
                    Function =>  GetInfo,
                    Abs     =>  'SQL_FN_NUM_ABS',
                    ACos    =>  'SQL_FN_NUM_ACOS',
                    ASin    =>  'SQL_FN_NUM_ASIN',
                    ATan    =>  'SQL_FN_NUM_ATAN',
                    ATan2   =>  'SQL_FN_NUM_ATAN2',
                    Ceiling =>  'SQL_FN_NUM_CEILING',
                    Cos     =>  'SQL_FN_NUM_COS',
                    Cot     =>  'SQL_FN_NUM_COT',
                    Degrees =>  'SQL_FN_NUM_DEGREES',
                    Exp     =>  'SQL_FN_NUM_EXP',
                    Floor   =>  'SQL_FN_NUM_FLOOR',
                    Log     =>  'SQL_FN_NUM_LOG',
                    Log10   =>  'SQL_FN_NUM_LOG10',
                    Mod     =>  'SQL_FN_NUM_MOD',
                    Pi      =>  'SQL_FN_NUM_PI',
                    Power   =>  'SQL_FN_NUM_POWER',
                    Radians =>  'SQL_FN_NUM_RADIANS',
                    Rand    =>  'SQL_FN_NUM_RAND',
                    Round   =>  'SQL_FN_NUM_ROUND',
                    Sign    =>  'SQL_FN_NUM_SIGN',
                    Sin     =>  'SQL_FN_NUM_SIN',
                    Sqrt    =>  'SQL_FN_NUM_SQRT',
                    Tan     =>  'SQL_FN_NUM_TAN',
                    Truncate    =>  'SQL_FN_NUM_TRUNCATE'
                    },
                "Cursor Scroll Concurrency"    =>  {
                    Desc        =>    "Concurrency  control  options  for
scrollable cursors",
                    Value   =>  'SQL_SCROLL_CONCURRENCY',
                    Function =>  GetInfo,
                    "Read Only" =>  'SQL_SCCO_READ_ONLY',
                    Lock    =>  'SQL_SCCO_LOCK',
                    "Optimistic    via    row    versions"            =>
'SQL_SCCO_OPT_ROWVER',
                    "Optimistic values" =>  'SQL_SCCO_OPT_VALUES'
                    },
                "Cursor Scroll Options"    =>  {
                    Desc    =>  "Scrolling options supported for cursors",
                    Value   =>  'SQL_SCROLL_OPTIONS',
                    Function =>  GetInfo,
                    "Forward Only"  =>  'SQL_SO_FORWARD_ONLY',
                    Static          =>  'SQL_SO_STATIC',
                    "Keyset Driven" =>  'SQL_SO_KEYSET_DRIVEN',
                    Dynamic         =>  'SQL_SO_DYNAMIC',
                    Mixed           =>  'SQL_SO_MIXED'
                    }
                );

$~ = "REPORT";
foreach $Attrib ( sort( keys( %Attributes ) ) )
{
    $Desc  = $Attributes{$Attrib}->{Desc};
    $Function = $Attributes{$Attrib}->{Function};
    $Value = eval("\$db->$Function(\$db->$Attributes{$Attrib}->{Value})");
    print "\n$Attrib  [$Attributes{$Attrib}->{Value}] = \"$Value\"\n";
    print "   $Desc.\n";

    foreach $Type ( sort( keys( %{$Attributes{$Attrib}} ) ) )
    {
        if( $Type eq "Desc" || $Type eq "Value" || $Type eq "Function" )
        {
            next;
        }

        undef $Const;
        undef $Supported;

        $Const = eval( "\$db->$Attributes{$Attrib}->{$Type}" );
        $Supported = ( $Value & $Const );
        if( $Supported > 0 )
        {
            $Supported = "Yes";
        }
        else
        {
            $Supported = "No";
        }
        write();
    }
}

print<<EOText;


Show how this ODBC Driver maps SQL Data Types:
----------------------------------------------
    The following table shows how this ODBC driver will map common
    SQL Data Types to it's ODBC Driver specific data type.
    it will also show the literal syntax, that is, how you deal with
    the data in an SQL statement. For example, if the literal syntax
    for the data type CHAR is 'DATA' then you would surround your
    CHAR data with single quotes as in :
        SELECT * FROM FOO WHERE Field1 = 'jello'
```

```
EOText

$~ = "Type_Header";
write();
$~ = "Type_Info";
foreach $Type ( ( @Types ) )
{
    if( $db->GetTypeInfo( eval( "\$db->SQL_$Type" ) ) )
    {
        undef %Data;
        if( $db->FetchRow() )
        {
            %Data = $db->DataHash();
            $Data{data_example}  =  $Data{LITERAL_PREFIX}  .  "DATA"  .
$Data{LITERAL_SUFFIX};
        }
        else
        {
            $Data{TYPE_NAME} = "---not supported---";
        }
        write();
    }
}

print "\n--== End Of Report ==--\n";
$db->Close();



format REPORT =
      @<<<<                                    @<<<<<<<<<<<<<<<<<<<<<<<<<<<
@<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
      $Supported, $Attributes{$Attrib}->{$Type}, $Type
.

format Type_Header =
    SQL Data Type     ODBC Driver Name    Literal Syntax
    ----------------- ------------------- --------------
.

format Type_Info =
    @<<<<<<<<<<<<<<<< @<<<<<<<<<<<<<<<<<< @<<<<<<<<
    $Type, $Data{TYPE_NAME}, $Data{data_example}
.
```

Odds and Ends

These are in no particular order. They are just a few little Perl programs that dealing with various things that really did not fit anywhere else.

Getting system memory information

This is a program written by Dave Roth. His website is http://www.roth.net/.

MEMORY.pl

```
use vars qw( $Log $Message $Name $Value );
use strict;
use Win32::OLE qw( EVENTS HRESULT in );

my @MEM_TYPES = qw(
  Unable_to_deterime Unknown Other DRAM
  Synchronous DRAM Cache DRAM EDO EDRAM
  VRAM SRAM RAM ROM Flash EEPROM FEPROM
  EPROM CDRAM 3DRAM SDRAM SGRAM
);

my @MEM_DETAIL = qw(
  Unable_to_deterime Reserved Other
  Unknown Fast-paged Static column
  Pseudo-static RAMBUS Synchronous
  CMOS EDO Window_DRAM Cache_DRAM
  Non-volatile
);

my @FORM_FACTOR = qw(
  Unknown_Type Non_Recognized_Type
  SIP DIP ZIP SOJ Proprietary_Type
  SIMM DIMM TSOP PGA RIMM SODIMM
);
```

```
my $Class = "Win32_PhysicalMemory";
my $Total;
my $iCount = 0;
(my $Machine = shift @ARGV || "." ) =~ s/^[\\\/]+//;
my            $WMIServices            =            Win32::OLE->GetObject(
"winmgmts:{impersonationLevel=impersonate,(security)}//$Machine/"   )   ||
die;


$~ = "INFO";
foreach my $Object ( in( $WMIServices->InstancesOf( $Class ) ) )
{
  my $Speed = $Object->{Speed} || "unknown speed";


  print    ++$iCount    .    ")    $Object->{Name}    ($FORM_FACTOR[$Object-
>{FormFactor}])\n";
  DumpInfo( "Tag", $Object->{Tag} );
  DumpInfo( "Type", $MEM_TYPES[$Object->{MemoryType}] );
  DumpInfo( "Detail", $MEM_DETAIL[$Object->{TypeDetail}] );
  DumpInfo( "Size", FormatMemory( $Object->{Capacity} ) . "bytes" );
  DumpInfo( "Speed", "$Speed (ns)" );
  DumpInfo(  "Location",  "$Object->{BankLabel}  ($Object->{DeviceLocator})"
);
  if( $Object->{DataWidth} != $Object->{TotalWidth} )
  {
    print "\tThis is ECC memory.\n";
  }
  $Total += $Object->{Capacity};
  print "\n";
}
printf( "\nTotal Memory: %s ( %sbytes )\n",
        FormatNumber( $Total ),
        FormatMemory( $Total ) );


sub FormatNumber
{
  my($Number) = @_;
  while( $Number =~ s/^(-?\d+)(\d{3})/$1,$2/ ){};
  return( $Number );
}

sub FormatMemory
{
  my( $Size ) = @_;
  my $Format;
  my $Suffix;
  my $K = 1024;
  my $M = $K * 1024;
  if( $M < $Size )
  {
    $Suffix = "M";
    $Format = $Size / $M;
  }
  elsif( $K < $Size )
  {
    $Suffix = "K";
    $Format = $Size / $K;
  }
  else
  {
    $Format = $Size;
  }
  $Format =~ s/\.(\d){1,2}\d+/.$1/;
  return( FormatNumber( $Format ) . " $Suffix" );
}

sub DumpInfo
{
  local( $Name ) = shift @_;
  local( $Value ) = shift @_;
  write;
}
```

```
format INFO =
        @<<<<<<<<<<<<<<<<<< ^<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
        "$Name:",              $Value
~~                          ^<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
                            $Value
.
```

Creating a file inside a Perl program

🖫 MOCKFILE.pl

```
use strict;

while (<DATA>)
{
  print $_ if $_ =~ /is/;
}

 __DATA__
This is a
file right
here in the
program.
No really
it is!
```

Using << to avoid a rash of print statements

In this example I have embedded some java code inside my Perl program. You can embed whatever you want. I just picked this. In an earlier example (ENFORM) we used this technique to really embed code we were going to run. Heck, if we wanted to get fancy we could write a Perl program that generated and compiled java code templates.

🖫 EMBEDDINGTEXTCODE.pl

```
use strict;

(my $list_statement = <<JAVACODE);

public class williclose extends javax.swing.JFrame {

    /** Creates new form williclose */
    public williclose() {
        initComponents();
    }

    /** This method is called from within the constructor to
     * initialize the form.
     * WARNING: Do NOT modify this code. The content of this method is
     * always regenerated by the Form Editor.
     */
    private void initComponents() {//GEN-BEGIN:initComponents
        jButton1 = new javax.swing.JButton();
        jLabel1 = new javax.swing.JLabel();

        addWindowListener(new java.awt.event.WindowAdapter() {
            public void windowClosing(java.awt.event.WindowEvent evt) {
                exitForm(evt);
            }
        });

        jButton1.setText("jButton1");
        getContentPane().add(jButton1, java.awt.BorderLayout.CENTER);

        jLabel1.setText("jLabel1");
        getContentPane().add(jLabel1, java.awt.BorderLayout.SOUTH);

        pack();
    }//GEN-END:initComponents
    /** Exit the Application */
    private   void   exitForm(java.awt.event.WindowEvent   evt)   {//GEN-
FIRST:event_exitForm
        System.exit(0);
    }//GEN-LAST:event_exitForm

    public static void main(String args[]) {
        new williclose().show();
    }



    // Variables declaration - do not modify//GEN-BEGIN:variables
    private javax.swing.JButton jButton1;
    private javax.swing.JLabel jLabel1;
    // End of variables declaration//GEN-END:variables

}
JAVACODE
print $list_statement;
```

This program will print all the line after the statement…

```
       (my $list_statement = <<JAVACODE);
```

…until the label JAVACODE.

Some XML (SAX)

XML is the buzzword today. I would be remiss if I did not cover this subject to some degree. I will spare you the sermon on XML, though I find it to be a remarkable and profound technology with regard to the transportation of data. We will be using the XML::Parser module written initially by Larry Wall and tweaked by Clark Cooper. While it is relatively easy to create xml "on the fly" in a Perl program this module provides some nice basic functionality for parsing xml. The module is object oriented so some explanation is in order less you believe in magic.

For my example, I have taken a fairly vanilla xml document that I created from another Perl program I wrote. Our goal is to use the XML::Parser module to parse this file and produce and html page.

I am assuming that you have a very basic understanding of xml and html. First, we will look at some the xml document, then take a look at the html page we create, then look at the code. I will then do a code walk through, and discuss the various parts of the program.

PS – SAX stands for "simple API for XML". Now you are buzzword compliant.

Our XML document to parse (a fragment)

🖫 PERLTOYCODE.xml

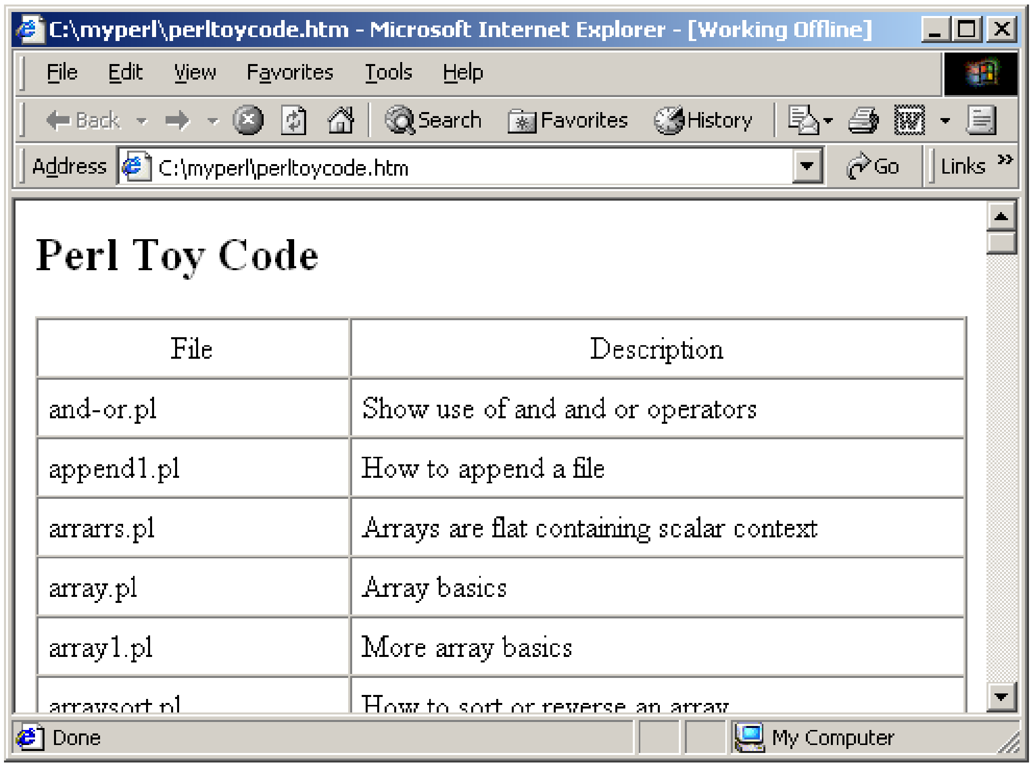
```
<codebase>

      <code><file>and-or.pl</file>

      <author>Eric Matthews</author>

      <description>Show use of andor operators</description>

      <use>Education, foundation</use></code>


      <code><file>append1.pl</file>

      <author>Eric Matthews</author>

      <description>How to append a file</description>

      <use>Education, foundation</use></code>
. . .
```

```
</codebase>
```

The results of our program

```
>perl –w parsexml1.pl
```

Running the program above produces the following html document.


C:\myperl\perltoycode.htm - Microsoft Internet Explorer - [Working Offline]
File Edit View Favorites Tools Help
Back Search Favorites History
Address C:\myperl\perltoycode.htm
Go Links
Done
My Computer

Perl Toy Code

File	Description
and-or.pl	Show use of and and or operators
append1.pl	How to append a file
arrarrs.pl	Arrays are flat containing scalar context
array.pl	Array basics
array1.pl	More array basics
arraysort.pl	How to sort or reverse an array

Our program code

PARSEXML1.pl

```
use strict;
use XML::Parser;
```

```
# instance xml parser
my $pXML = new XML::Parser();

# set call functions
$pXML->setHandlers(
                  Start => \&find_begin_tag,
                  End => \&find_end_tag,
                  Char => \&get_data
                 );

# tag we want to process
my $TagToProcess = "";

# dump results to file
open (STDOUT, ">perltoycode.htm");

# set up HTML page
print "<html><head></head><body>";
print "<h2>Perl Toy Code</h2>";
print "<table border=1 cellspacing=0 cellpadding=5>";
print "<tr>
            <td align=center>File</td>
            <td align=center>Description</td>
       </tr>
      ";

# parse xml file
$pXML->parsefile("perltoycode.xml");

print "</table></body></html>";

# called when start tag is found
sub find_begin_tag()
{
  my ($parser, $name, %attr) = @_;
  $TagToProcess = lc($name);

  print "<tr>" if $TagToProcess eq "code";
    print "<td>" if $TagToProcess eq "file";
    print "<td>" if $TagToProcess eq "description";
}

# called if tag data is found
sub get_data()
{
  my ($parser, $data) = @_;

  print $data if $TagToProcess eq "file";
  print $data if $TagToProcess eq "description";
}

# called if end tag is found
sub find_end_tag()
{
  my ($parser, $name) = @_;
  $TagToProcess = lc($name);

  print "</tr>" if $TagToProcess eq "code";
    print "</td>" if $TagToProcess eq "file";
    print "</td>" if $TagToProcess eq "description";

  $TagToProcess = "";
}
```

Code walk through

Parsing an xml document boils down to node processing. Our goal is to read the document finding all the <file> </file> and <description> </description> tag pairs. Our goal is to extract these tags and create an html table. I am only going to cover the code directly pertaining to parsing the xml document.

Creating an instance of the parser

```
use XML::Parser;

my $pXML = new XML::Parser();
```

The second statement also serves to intialize the parser object. $pXML is a reference to the parser object.

ParseFile( ) Method

This is the method we use to parse our xml file. It is a pretty straight forward affair.

Syntax

```
$<ref_to_parser_object>-parsefile("<xml_file_to_parse>");
```

Example

```
$pXML->parsefile("perltoycode.xml");
```

Setting the handlers

In parsing our xml document there are really three related entities that we need to be on the lookout for. They are:

- ☒ The desired opening tag
- ☒ The desired closing tag
- ☒ The data contained within the opening and closing tag

To deal with these three entities we create three handlers. It just so happens the module has three boilerplate entities for these entities. In order to create our handlers we use the setHandlers( ) method. This method expects a hash of key/value pairs. Lets have a look at the code.

```
$pXML->setHandlers(

                  Start => \&find_begin_tag,

                  End => \&find_end_tag,

                  Char => \&get_data

                 );
```

The key is the method we want to use in the module. The value will represent the callb ack to a subroutine in our program that will contain the code that in concert with its associated method represents the "event handler" or more specifically what we want to do when we encounter a given entity.

If you are not all that solvent in object oriented programming this may seem to be a bit like magic. Personally, I do not think it is all that important understand the technical details of how this works. What you need to be aware of at a high level is that we need a means of triggering an event as we parse the xml document. We want to trigger an event when we hit the desired opening tag, closing tag, and data contained within the tag so we can write the appropriate html code (in this case).

Please make note that there are more handlers available than we are covering here. I debated whether to discuss the implementation of how the module handles the relationship between its method and your subroutine but thought that it would just take us too far astray from our objective.

Also, the whole point of reuse is not to have to get bogged down by the implementation details so we are just not going down that road. What you need to understand is how to define handlers for these three methods.

Now that we have done that it is time to look at our three related subroutines.

Our opening tag subroutine

Remember, when we define our handler we created a reference to our sub for the Start( ) tag method.

```
Start => \&find_begin_tag,
```

Here is the code for the find_begin_tag( ) subroutine.

```
sub find_begin_tag()

{

  my ($parser, $name, %attr) = @_;



  print "<tr>" if $TagToProcess eq "code";

    print "<td>" if $TagToProcess eq "file";
    print "<td>" if $TagToProcess eq "description";

}
```

Assume for the moment that as the xml document is being parsed the "chunk" that represent the opening tag is stored in the array @_. @_ holds three elements. The first element is the reference to the parser object. The second element is the name of the xml element (tag name). Finally, if the element contains attributes they will be returned in a hash, otherwise the hash will be undefined. Notice we assign @_ to our list with the statement.

```
my ($parser, $name, %attr) = @_;
```

Then, since it is the element name I care about I assign it to a scalar.

```
$TagToProcess = lc($name);
```

I am using the lowercase function to downshift the element name. I am doing this to deal with documents that use mixed case, so we can enforce some semblance of consistency and continuity, and protect ourselves from a case of O'Brian's law.

I then analyze the tag and if it is one of the ones I want I process it.

```
  print "<tr>" if $TagToProcess eq "code";

    print "<td>" if $TagToProcess eq "file";

    print "<td>" if $TagToProcess eq "description";
```

As I said earlier, I am assuming some basic understanding of html on your part. This said, the code above should make sense to you.

Our closing tag subroutine

The nice thing about this is that once we have seen once subroutine the others are pretty much the same. About the only real difference is with respect to the context of what part of the xml document we are currently parsing.

The closing tag will only contain two elements in the returned array @_. The parser handle, and the closing tag name.

Our subroutine for dealing with the data

The closing tag will only contain two elements in the returned array @_. The parser handle, and CDATA (which is the data). We certainly can apply other html formatting to our data.

```
  print "<b>" . $data . "</b>" if $TagToProcess eq "file";
```

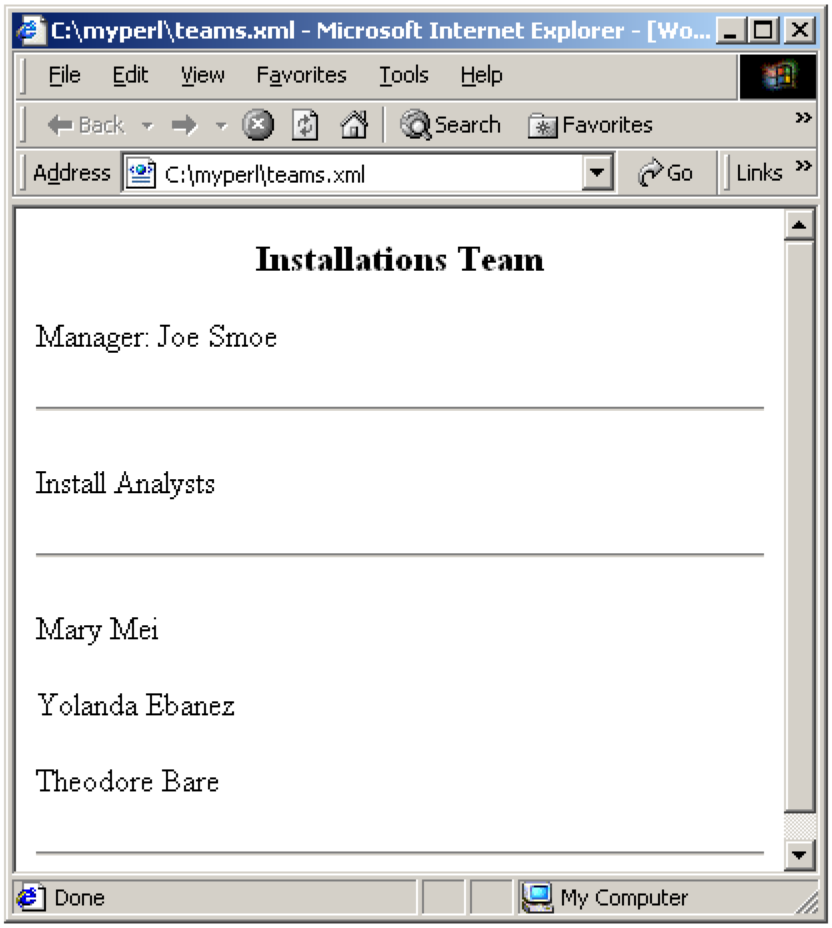
Parsing a more complex XML document

I wanted to demonstrate a more complex example of parsing xml. Here is the xml document.

TEAMS.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml:stylesheet type='text/xsl' href='teams1.xsl'?>

<teams>
      <Installations>
            <Manager>Joe Smoe</Manager>
            <Install_Analyst>Mary Mei</Install_Analyst>
            <Install_Analyst>Yolanda Ebanez</Install_Analyst>
            <Configuration_Specialist>William Overdue
            </Configuration_Specialist>
            <Install_Analyst>Theodore Bare</Install_Analyst>
      </Installations>

      <Development>
            <Manager>Anne Boolean</Manager>
            <Designer>Tim Thumb</Designer>
            <Designer>Tim Thumb</Designer>
            <Programmer>Melvin Atonin</Programmer>
            <Programmer>Mitchell Ray</Programmer>
            <Tester>Sam Selino</Tester>
            <Architect>Dorothy Oz</Architect>
            <Programmer>Tammy Smith</Programmer>
      </Development>
</teams>
```

This document will render in the browser as follows because of it's associated XSL stylesheet.


C:\myperl\teams.xml - Microsoft Internet Explorer - [Wo...
File Edit View Favorites Tools Help
Back Search Favorites
Address C:\myperl\teams.xml Go Links
Done My Computer

Installations Team

Manager: Joe Smoe

Install Analysts

Mary Mei

Yolanda Ebanez

Theodore Bare

This works fine for browsers that can handle XML and XSL. We can use the XML parser to create an HTML document that does the same thing.

You will see that the code is not that much different than in the last example.

PARSEXML2.pl

```
use strict;
use XML::Parser;

my $ok="y";

my %StartTag = (
         "installations" =>"<hr><h2>Installations Team</h2><ul>",
          "development" => "<hr><h2>Development Team</h2><ul>",
          "manager" => "<h3><b><hr>",
          "install_analyst" => "<p>",
          "designer" => "<p>",
          "programmer" => "<p>",
          "architect" => "<p>",
          "tester" => "<p>",
          "configuration_specialist" => "<p>"
               );

my %EndTag = (
    "installations" => "</ul>",
    "development" => "</ul>",
    "manager" => " - Manager</h3></b>",
    "install_analyst" => " - Install Analyst</p>",
    "designer" => " - Designer</p>",
    "programmer" => " - Programmer</p>",
    "architect" => " - Architect</p>",
    "tester" => " - Tester</p>",
    "configuration_specialist" => "- Configuration Specialist</p>"
             );



# name of XML file
my $file = "teams.xml";

# initialize parser
my $pXML = new XML::Parser();

$pXML->setHandlers(
                   Start => \&find_begin_tag,
                   End => \&find_end_tag,
                   Char => \&get_data,
                   Final => \&done_parsing
                   );

open (STDOUT, ">teams.htm");

print "<html><head></head><body>";

$pXML->parsefile($file);

print "</body></html>";

my $sTag;
sub find_begin_tag() {
      my ($parser, $name, %attr) = @_;
      $name = lc($name);
      print $StartTag{$name} if ($StartTag{$name} and $ok);
      $sTag = $StartTag{$name}
}

sub get_data() {
      my ($parser, $data) = @_;
      print $data if ($sTag and $ok);
}

sub find_end_tag() {
      my ($parser, $name) = @_;
      $name = lc($name);
      print $EndTag{$name} if $EndTag{$name};
      $sTag = "";
}
```

In this example I want to point out the following lines of code.

```
print $StartTag{$name} if ($StartTag{$name}
```

…and…

```
print $EndTag{$name}
```

Here we are using the callback function with the reference to our hashes. The callback function will perform for each key in our key.

Some XML (DOM)

Six of one, half dozen of another seems to also be no stranger to the world of XML. Actually I would be hard pressed to find another technology that has more acronyms. I have looked at a number of sources that show me how to do the same tasks with both DOM and SAX. I generally walk away from these experiences thinking, "and the point would be". I am going to try to stick to showing you stuff in DOM that I have not shown you how to do with SAX. Oh, DOM stands for document object model.

The DOM parser is actually derived from the SAX parser. I do not think the technical details of this are all that important though. Originally I was not planning on showing DOM but found SAX not all that helpful in easily extracting a single node. I have also provided a few other examples for your amusement.

What I have covered regarding XML parsing is by no means complete. I have ignored dealing with attributes at this time as it is not clear to me what our XML strategy and architecture will fully entail. Yes, this is my story and I am sticking to it. You will want to get a hold of the documentation for this module. It is my hope that this section will give you enough context for using the module that you will be able to do further exploration on your own.

Finally, I am assuming a pretty reasonable familiarity with XML on your part. Onward.

Node extraction

TEAMS.xml

Consider the following XML document.

```
<?xml version="1.0" encoding="UTF-8"?>
<teams>
      <Installations>
            <Manager>Joe Smoe</Manager>
            <Install_Analyst>Mary Mei</Install_Analyst>
            <Install_Analyst>Yolanda Ebanez</Install_Analyst>
            <Configuration_Specialist>William Overdue
            </Configuration_Specialist>
            <Install_Analyst>Theodore Bare</Install_Analyst>
      </Installations>
      <Development>
            <Manager>Anne Boolean</Manager>
```

```
             <Designer>Tim Thumb</Designer>
             <Designer>Tim Thumb</Designer>
             <Programmer>Melvin Atonin</Programmer>
             <Programmer>Mitchell Ray</Programmer>
             <Tester>Sam Selino</Tester>
             <Architect>Dorothy Oz</Architect>
             <Programmer>Tammy Smith</Programmer>
      </Development>
</teams>
```

I want to only extract the 'Installations' node from the document. This action is a bit problematic using SAX as we also end up getting the 'Manager' tag from the 'Development' node. I think DOM offers a much more elegant means of accomplishing this task. The code.

DOM-EXTRACT_NODE.pl

```
use strict;
extsectUsage() unless @ARGV >= 2;

sub extsectUsage {
    die <<'USAGE';
Perl script to extract named sections from a file.

Usage:
    perl dom-extract_node.pl <xml_file> <node_to_extract>

Example, putting output into a file:
    perl dom-extract_node.pl teams.xml Development
USAGE
}

use XML::DOM;

my $file = $ARGV[0];  # what xml file?
my $extract_node = $ARGV[1]; # node we want to extract
my $xmlP = new XML::DOM::Parser();
my $Xdoc = $xmlP->parsefile($file);
my $root = $Xdoc->getDocumentElement();
my @nodes = $root->getChildNodes();
my @kids; # to get the children nodes

# children nodes to extract if present
my @occup = (
             "Manager",
             "Install_Analyst",
             "Configuration_Specialist",
             "Designer",
             "Programmer",
             "Tester",
             "Architect"
            );

extract_node();

sub extract_node
{
# an evil triple nested loop
 foreach my $node (@nodes)
 {
   if ( $node->getNodeName() eq $extract_node ) # node to extract
   {
     @kids = $node->getChildNodes(); # get list of children nodes
     foreach my $elem (@kids) # loop through kids
     {
       for  (my  $i=0;  $i<@occup;  $i++)  #  looking  to  extract  nodes  that
match element in @occup
       {
         print $elem->getFirstChild()->getData
              . "\n"
              if $elem->getNodeName() eq $occup[$i];
            }
        }
      }
  }
}
```

You need to install the XML::DOM module if you have not already not done so.

Create instance of DOM parser object

```
my $xmlP = new XML::DOM::Parser();
```

Create scalar references to various method calls

```
my $Xdoc = $xmlP->parsefile($file);
```

The parsefile( ) method is used to parse XML files. It is possible to embed XML in a Perl program. To parse embedded XML you use the parse( ) method. DO NOT get these two methods confused or you will be quite frustrated.

```
my $root = $Xdoc->getDocumentElement();
```

The getDocumentElement() method is a handy little function that allows us direct access to the child node that is the root element of the document. Here is an example of how this is being utilized in our program.

You can see we are using $root to call the getChildNodes() method which will return a list of all the child nodes to our array.

```
my @nodes = $root->getChildNodes();
```

We have also created an array to specifiy the list of occupations we want to extract. Extraction becomes a reference nightmare. I am not even going to attempt to cover this in this text. I have coded a series of nested loops to extract the data. This loop structure is part of the subroutine named extract_node( ).

```
foreach my $node (@nodes)
```

```
{
   if ( $node->getNodeName() eq $extract_node )  {
     @kids = $node->getChildNodes();
        foreach my $elem (@kids)
        {
            for (my $i=0; $i<@occup; $i++)
            {
         print
                 $elem->getFirstChild()->getData
               .  " - "
               .  $elem->getNodeName()
               . "\n"
              if $elem->getNodeName() eq $occup[$i];
            }
       }
}
```

In the outer most loop we are looking for the node that we want to extract. Once we find this loop we populate the array @kids by calling the method getChildNodes(). Note that we are now calling this method using the reference $node. Remember up above we used getChildNodes() to get all the children that are hierarchically under the root node. Here we are looping through this array of references until we find the desired one.

```
foreach my $node (@nodes)
if ( $node->getNodeName() eq $extract_node )
     @kids = $node->getChildNodes();
```

Once we have accomplished this we then want to look through this list looking for all the nodes that match our occupation array. We need a loop to process @kids.

```
        foreach my $elem (@kids) {
```

And, we need an inner loop to see if each element of @kids matches one of our specified occupations that we defined in the array @occup.

```
            for (my $i=0; $i<@occup; $i++)

            {

         print

                 $elem->getFirstChild()->getData

              .  " - "

              .  $elem->getNodeName()

              . "\n"

               if $elem->getNodeName() eq $occup[$i];            }
```

If we get a match we print the name out. As you can see traversing an XML document can be quite a ghoulish affair.

toString method

It is possible once you open an XML document to dump its contents to a string. Of course, needless to say, you do not need XML::DOM to accomplish this task. The following program shows use of this method. The program opens and XML document using the parser and writes the document to another file. It does it in whole, but is meant to illustrate that you could use these functions to parse and XML document and write only desired nodes to another XML document.

DOM_TOSTRING.pl

```
use strict;
use XML::DOM;


# create an XML-compliant string
```

```
my $xmlfil = "teams.xml";

my $xmlDP = new XML::DOM::Parser();

# create parse tree in memory
my $doc = $xmlDP->parsefile($xmlfil);

# dump tree as string to file
open (STDOUT, ">dom-a.xml");
print $doc->toString();
```

printToFile method

🖫 DOM_PRINTTOFILE.pl

```
use strict;
use XML::DOM;

my $iFil = $ARGV[0];
my $p = XML::DOM::Parser->new;
my $doc = $p->parsefile($iFil);

my $nodes = $doc->getElementsByTagName ("CODEBASE");
my $n = $nodes->getLength;
print $n . "\n";

for (my $i = 0; $i < $n; $i++)
{
  my $node = $nodes->item ($i);
  my $href = $node->getAttribute ("HREF");
}

#extract filename less extension
$iFil = $1 if $iFil =~ /^(.+)\./ ;
```

```
#filename concat '-copy', concat extension
$iFil = $iFil . "-copy" . ".xml";


$doc->printToFile ($iFil);  #create copy of XML doc
```

Embedded XML

As I said earlier, it is possible to embed an XML in a Perl program. Here is an example.

🖫 DOM_EMBEDXML.pl

```
use strict;
use XML::DOM;


# create xml string
      # Note, you must escape "
my $xmlstr  = "<?xml version=\"1.0\"?>
               <teams>
             <Installations>
            <Manager>Joe Smoe</Manager>
            <Install_Analyst>Mary Mei</Install_Analyst>
            <Install_Analyst>Yolanda Ebanez</Install_Analyst>
            <Configuration_Specialist>William
Overdue</Configuration_Specialist>
            <Install_Analyst>Theodore Bare</Install_Analyst>
             </Installations>


             <Development>
            <Manager>Anne Boolean</Manager>
            <Designer>Tim Thumb</Designer>
            <Designer>Tim Thumb</Designer>
            <Programmer>Melvin Atonin</Programmer>
            <Programmer>Mitchell Ray</Programmer>
            <Tester>Sam Selino</Tester>
            <Architect>Dorothy Oz</Architect>
            <Programmer>Tammy Smith</Programmer>
```

```
              </Development>
              </teams>


              ";
my $xmlDP = new XML::DOM::Parser();
# create parse tree in memory
my $doc = $xmlDP->parse($xmlstr);
print $doc->toString();
```

There are only two items I want to point out from this example. The first is pertaining to the XML. Notice that I have escaped the quote marks.

```
my $xmlstr  = "<?xml version=\"1.0\"?> …
```

You must to this (or an equivalent of this) or your program will not work. Second, notice we are using the parse( ) method instread of the parsefile( ) method.

```
my $doc = $xmlDP->parse($xmlstr);
```

Get XML version

This is a simple program that demonstrates the use of the getVersion( ) method which may come in handy one day when there are seven bazillion version of XML.

```
<?xml version="1.0" encoding="UTF-8"?>
```

DOM_GETVERSION.pl

```
use strict;
use XML::DOM;
my $iFil = $ARGV[0];
my $xmlDP = new XML::DOM::Parser();
my $doc = $xmlDP->parsefile($iFil);
# get XML PI
my $decl = $doc->getXMLDecl();
# get XML version
```

```
print "Version: " . $decl->getVersion() . "\n";
```

This is all for now.

Some CGI

First, let me say that this will not be a complete dissertation on using the CGI module. I first want to call your attention to the documentation for this module. It is excellent, and provides a wealth of examples. The author, Lincoln Stein, also wrote a book named, "Using CGI" which in my opinion is the best book on the market for this subject. I would encourage you to buy it if you are planning on doing a lot of CGI programming. If you are looking for a more "general purpose" programming book that has good CGI coverage, check-out "The Perl Black Book" by Steven Holtzner.

This section assumes that you have a web server. You can use any web server that supports Perl. Microsoft's PWS (Personal Web Server) or Apache are good choices, and both can be had for free.

I also need to assume that you understand html. I will be covering this subject with the idea that you have written html before. If you have only created web pages through use of a tool like Front Page then you may have some trouble with these materials.

It is my intent to show four examples. These examples will give pretty good coverage to this subject. Since these programs are rather large I will not be listing the entire code in this document.

Creating a CGI form

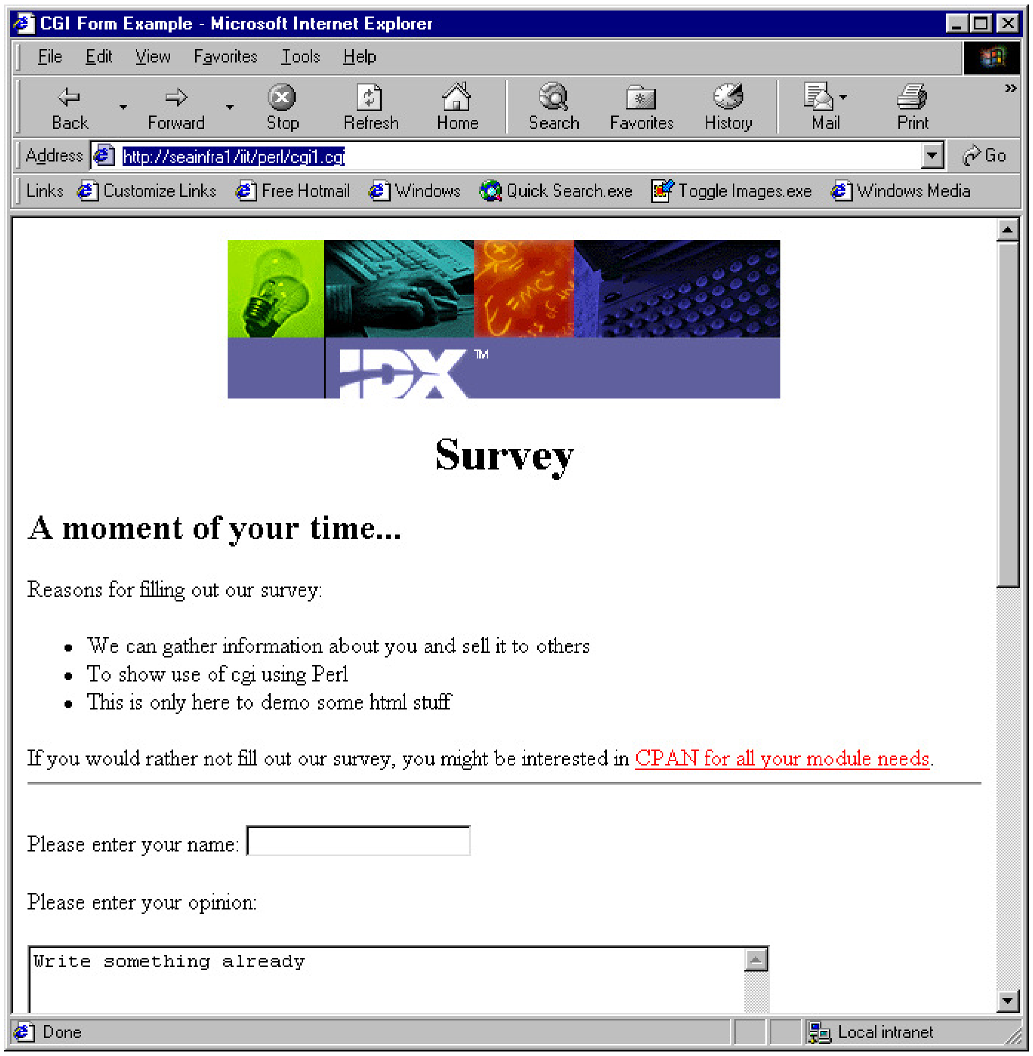
Perl CGI programming allows you create dynamic html. And, sometimes dynamic is just better than static! I am not going to head off into any sermons or try to start any holy wars over whether you should use ASP, or Java Servlets, or Perl CGI for creating dynamic web content. I have used all three. I am actually quite fond of ASP. But, this is a Perl document so Perl is what we shall cover.

🖫 CGI1.cgi

This is an example cgi script that creates an html form. If you are within our casino domain you can access this code via the following url.

http://seainfra1/iit/perl/cgi1.cgi

You can also run this and the other cgi scripts from your web server. I will not be covering how to set up and configure a web server as part of this document. Before we start examining the code I have included a screen shoot of the web page produced by this script.


CGI Form Example - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search Favorites History Mail Print
Address http://seainfra1/iit/perl/cgi1.cgi
Go
Links Customize Links Free Hotmail Windows Quick Search.exe Toggle Images.exe Windows Media
Done
Local intranet

Survey

A moment of your time...

Reasons for filling out our survey:

- We can gather information about you and sell it to others
- To show use of cgi using Perl
- This is only here to demo some html stuff

If you would rather not fill out our survey, you might be interested in CPAN for all your module needs.

Please enter your name:

Please enter your opinion:

Write something already

It might be a good idea if you have a copy of the source at your fingertips.

Creating a CGI object

To create a CGI object you just assign a scalar to "new CGI". The scalar then becomes our reference that we will be using to create our dynamic html.

```
$co = new CGI;
```

Creating the HTML page

Notice how we use $co and the infix operator to reference the various methods and members of the CGI module. “start_html” is the equivalent of the <html>tags and other related tags and properties. I am leaving it up to you to associate these to the rendered HTML.

```
$co->start_html

(

    -title=>'CGI Form Example',

    -author=>'EM',

    -meta=>{'keywords'=>'CGI Perl'},

    -BGCOLOR=>'white',

    -LINK=>'red'

),
```

```
$co->end_html;
```

Creating an HTML form

```
$co->start_form

(

    -method=>'POST',

    -action=>"http://seainfra1/iit/perl/cgi2.cgi"

),
```

The action represents what we want to happen when the user clicks the submit button. Our method is set to ‘POST’. As you can see, our intent is to call another CGI script.

```
$co->end_form,
```

Creating Form Objects

Here is some code for creating HTML form objects, etc…

Images

```
$co->center($co->img({-src=>'idx.bmp'})),
```

Heading Tags

H1-H6

```
$co->center($co->h1('Survey')),
```

Paragraph Tag

```
$co->p,

$co->p($mytext),
```

Text box

With Label:

```
"Please enter your name: ",

$co->textfield('text'), $co->p,
```

Radio Buttons

```
$co->radio_group

(

    -name=>'radios',

    -values=>['1','2','3', '4', '5', '6', '7'],

    -default=>'1',

    -labels=>\%labels

),
```

```
$labels{'1'} = 'Sunday';

$labels{'2'} = 'Monday';
```

```
$labels{'3'} = 'Tuesday';

$labels{'4'} = 'Wednesday';

$labels{'5'} = 'Thursday';

$labels{'6'} = 'Friday';

$labels{'7'} = 'Saturday';
```

Unordered List

```
$co->ul

(

    $co->li('We can gather information about you and sell it to
others'),

    $co->li('To show use of cgi using Perl'),

    $co->li('This is only here to demo some html stuff'),

),
```

Hyperlinks

```
$co->a({href=>"http://www.cpan.org/"},"CPAN for all your module
needs"), ".",
```

Check boxes

```
$co->checkbox_group

(

    -name=>'checkboxes',

    -values=>['VB','Perl','C/C++','Cobol','Ada','Java','Tacl'],

    -defaults=>['Bread','Cruise missles']

),
```

Horizontal Rule (a line)

```
$co->hr,
```

List box

```
$co->popup_menu
(
    -name=>'popupmenu',
    -values=>['Very much','A lot','As much as you can send','Tons']
),
```

Hidden text field

```
$co->hidden
(
    -name=>'hiddendata',
    -default=>'big brother is watching you'
),
```

Text Area

```
$co->textarea
(
    -name=>'textarea',
    -default=>'Write something already',
    -rows=>10,
    -columns=>60
),
```

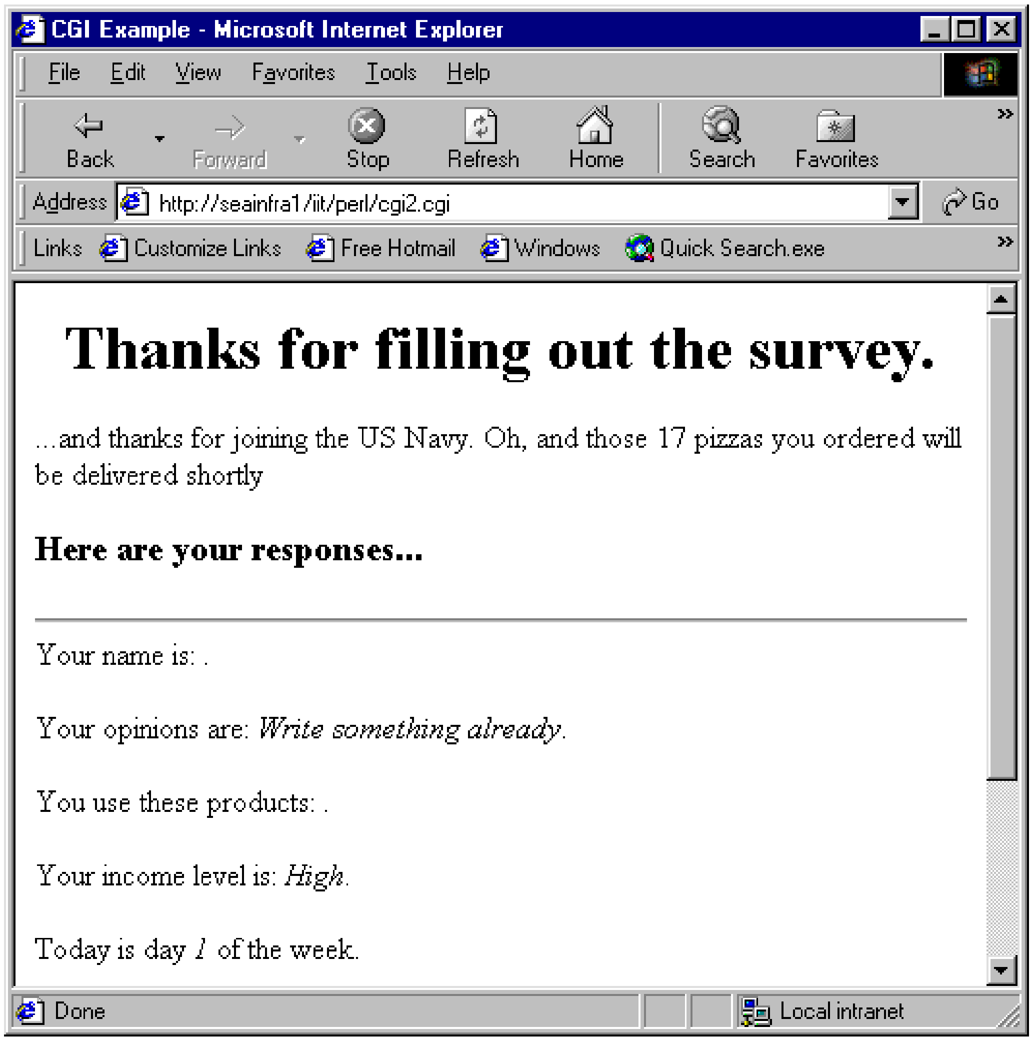
Password field

```
$co->password_field
(
    -name=>'password',
    -default=>'maryquitecontrary',
    -size=>25,
),
```

The second CGI script echos back what the user entered. More than likely you would take the user input and stash it in some database, though certainly for things likes orders you would want to have the person verify the information they have submitted.

Getting HTTP parameters

Let's look at the second CGI programs code and then I will discuss what transpired, and draw you a picture or two.


CGI Example - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search Favorites
Address http://seainfra1/iit/perl/cgi2.cgi
Go
Links Customize Links Free Hotmail Windows Quick Search.exe
Done
Local intranet

Thanks for filling out the survey.

...and thanks for joining the US Navy. Oh, and those 17 pizzas you ordered will be delivered shortly

Here are your responses...

Your name is: .

Your opinions are: *Write something already*.

You use these products: .

Your income level is: *High*.

Today is day *1* of the week.

CGI2.cgi

There is not really any new stuff covered here. The "million dollar" question is how did we get the data from the other CGI script?

```
if ($co->param()) {

    print

        "Your name is: ",$co->em($co->param('text')),

        ".",

        $co->p,


 "Your opinions are: ",$co->em($co->param('textarea')),

        ".",

        $co->p,


        "You use these products: ",$co->em(join(", ",

        $co->param('checkboxes'))), ".",

        $co->p,


"Your income level is: ",$co->em($co->param('list')),

        ".",

        $co->p,


        "Today is day ", $co->em($co->param('radios')),

        " of the week.",

        $co->p,
"Your password is: ",$co->em($co->param('password')),

        ".",

        $co->p,


        "How much unsolicited mail you like: ",

        $co->em($co->param('popupmenu')),

        ".",

        $co->p,


        "The hidden data is ",$co->em(join(", ",

        $co->param('hiddendata'))),

      ".";

}
```

We make a call to the param( ) method. If we received parameters as part of the HTTP transmission (in the HTTP header), then this method will be true. Here is how we get the data.

```
$co->em($co->param('text')),
```

param( ) accepts as an argument the name of the HTTP parameter. In the case of this example that would be name of the text box from CGI1.cgi. The code we used in the previous program was:

```
$co->textfield('text'), $co->p,
```

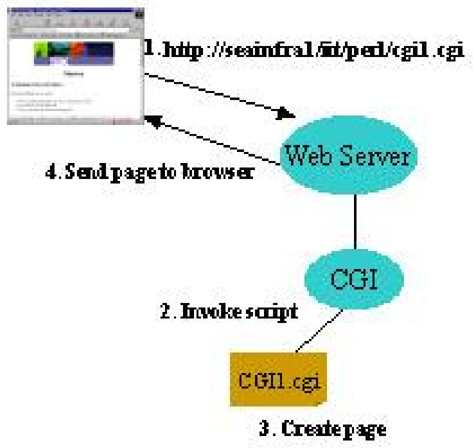
We could just have easily passed the parameter as part of the url by saying…

http://seainfra1/iit/perl/cgi1.cgi**?text="foobar"**

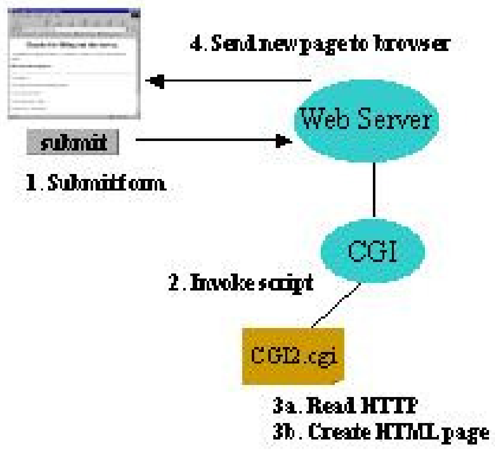
PS- 'em' means place what follows in italics.

I also want to note that these two programs are entirely separate from one another. There is no context maintained between them. CGI1.cgi submits to CGI2.cgi passing it all of the parameters from the form as part of an HTTP string.

Visually this is what is happening.



A request is made for url http://seainfra1/iit/perl/cgi1.cgi . The web server gets and executes this script. This script produces an html form that is delivered back to the browser.



The user fills out the form, and submits it back to the web server. "cgi2.cgi" is invoked as part of the submit method on the form. This program is executed by the web server. Its purpose is to read the responses from the form that was submitted and create another HTML page that is sent back to the browser with the users responses.

Another example

If you have been to my intranet web site you may have used my search engine for the hundreds of Perl scripts I (and others) have written. This engine was created using Perl and CGI. I am showing it to you for a couple of reasons. One, it recreates the same HTML page each time a selection is made. The other reason is that it interacts with the file system on the server. This program opens and reads 200+ programs, creating a dynamic web page on the fly. It does this in a second or two. I wanted to show you that Perl CGI can be pretty robust.

This program is 250+ lines of code, so I will only be showing certain sections.

```
use CGI::Carp qw/fatalsToBrowser/;
```

The above line of code will cause cgi proram execution errors to be displayed in the browser. This is probably something you always want to include as a default to help assist you in trouble shooting.

The high level flow of this script is as follows.

1. Get a list of all the files in the perl code directory that have the extension .pl
2. Open each file and extract the index (and other) tags.
3. Produce an HTML page based on the users index selection

This CGI script is actually more conventional Perl text processing than it is CGI. I will leave it up to you to review the code. I merely wanted to offer it up to you as an example

Using Telnet with CGI

This script makes a connection to our Compaq NonStop system named \Snoball, executes an Entrude query and formats the results as an HTML page. This will give you a sense of what you can do with CGI and Telnet. I kept this example very simple and bare bones. For more robust examples you can visit http://seainfra1/dba/, and take a look at the database tools that are available.

TELNET.cgi

```
use Net::Telnet;
 use CGI qw(:standard);
 my $Command;
 my $output;

# $Option = param("Option");
# $Param = param("Param");

 $Command = param("Command");

 if ($Command eq "") { $Command = "files"; }

 $telnet = new Net::Telnet (Timeout=>30, Errmode=>'die');

 print "HTTP/1.0 200 OK\n";
 print "Content-type: text/html\n\n";

 print      "<link      rel='stylesheet'      href='/dba/scripts/dba.css'
type='text/css'>";
 $output= $telnet->open('snoball.idx.com');
 $telnet->waitfor('/Enter Choice>/i');
 $telnet->print('TACL');
 $telnet->waitfor('/TACL 1>/i');
 $telnet->print('LOGON S5.SL');
 $telnet->waitfor('/Password/i');
 $telnet->print('');
 $telnet->waitfor('/\$D13.SLOBEY 1>/i');
 $telnet->print('setprompt none');
 $telnet->waitfor('/[0-9]+\>/i');
 print  "<H2>Connected  to  SNOBALL.  Results  of  command  $Command  listed
below...</H2>\n<PRE>";

# $telnet->print('inlprefix +');
# $telnet->print('entruder/inline/');
# $telnet->print('+ open reg-rec;');
# $telnet->print('+ show;');
# $telnet->print('+ exit;');
# $telnet->print('inleof');

 $telnet->print($Command);
 print $telnet->waitfor('/[0-9]+\>/i');

 $telnet->print('bye');
 print "<b>Done.</B></PRE></BODY></HTML>";
exit;
```

This should be enough to get you going with Perl and CGI.

Reading a binary File

Here is an example on how to read binary data. There are not a lot of good examples out there. I think one of the reasons for this has to do with the myriad of ways that integers get implemented on systems. There is a host other reasons I will not bore you with. The goal here is to show a practical toy code example. I have chosen a Compaq NonStop Enscribe database file as my example because that is where I work and what many folks have to deal with. You should get the general gist and twist from this example.

A basic example

READREGID.pl

```
use strict;
my $record;
my $RECSZ=32;
my @fields;
my @field_names = ("MRN",
                   "IDNUM-FAC",
                   "PTPTR",
                   "PTPTR-FAC",
                   "ADD-DATE",
                   "ADD-TIME",
                   "ADD-USER");

open (FILE, "regid") or die $!;
binmode FILE;
while (<FILE>)
{
  read (FILE, $record, $RECSZ);
  @fields = unpack("A8iNnnnnA10",$record) ;
  print_recs();
}
my $ii=1;
sub print_recs
```

```
{
  for (my $i=0; $i<@fields; $i++)
  {
     print "$field_names[$i]:  $fields[$i] \n" if $i < 7;
     print "\n" if $i == 6;
     exit if ++$ii > 23;
  }
}
```

Before we get into the code lets first discuss what the program is doing. I am reading a structured database file named REGID. Each record in this file is 32 bytes and the schema is as follows.

Field name	Data type	Precision
MRN	Character	8 bytes
IDNUM-FAC	Integer	2 bytes
PTPTR	Integer	4 bytes
PTPTR-FAC	Integer	2 bytes
REGID-ADD-DATE	Integer	2 bytes
REGID-ADD-TIME	Integer	2 bytes
REGID-ADD-USER	Integer	2 bytes
FILLER	Character	10 bytes

Records are fixed length and in this data definition language the word 'Filler' is a keyword that is used to denote space reserved for future expansion. In other words, we might one day want to add some fields to our record so we reserve some space using by using filler.

In the program it is our goal to read the record and print both the column/field name and the data. We will do this for the first three records.

NOTE: I would recommend that you use a query tool or program that has the ability to access database data via an index. This is a brute force method for reading binary data and is not all that efficient with respect to structured database files.

Now for the code…

I am using four variables…

```
my $record;

my $RECSZ=32;

my @fields;

my @field_names
```

- $record is the scalar that I will use to hold a record.
- $RECSZ is a scalar used to hold the length of a record.
- @fields is an array that I use to capture data from each field
- @field_names is an array that holds the field names.

You could certainly enhance this program by programmatically generating the @field_names array by reading the schema file/code for the file.

I also have set our filehandle to binmode. This is necessary in DOS and Windows because these OS's differentiate between binary and text files. In these environments the \cM\Cj end-of-line markers get translated into newlines (\n). If this sequence appears in a binary file it needs to be left alone! Bottom line here is that regardless of the OS you are running your Perl program you can leave the following line and it will be just fine (and make your code more platform independent)

```
binmode FILE;
```

I open the file just like I would a text file, and create a while loop just like I would for reading the text file. Instead of reading $_, I need to read the file so many bytes as a time. I do this by using the read( ) function. This function accepts three arguments: the filehandle, a variable to hold what we read, and the chunk we want to read.

```
  read (FILE, $record, $RECSZ);
```

What ends up in our scalar $record is a mixture of textual data and integer data that we need to convert. We need to further dissect each record down to the field level. To do these we create a template using the unpack( ) function.

unpack takes two arguments: the template, and a list (in our case the scalar that is the record itself…a lit with one item).

```
  @fields = unpack("A8iNnnnnA10",$record) ;
```

The template is a bit cryptic. The table below will help clarify how it all works.

Field name	Data type	Precision	Template
MRN	Character	8 bytes	**A8**
IDNUM-FAC	Integer	2 bytes	**n**
PTPTR	Integer	4 bytes	**N**
PTPTR-FAC	Integer	2 bytes	**n**
REGID-ADD-DATE	Integer	2 bytes	**n**
REGID-ADD-TIME	Integer	2 bytes	**n**
REGID-ADD-USER	Integer	2 bytes	**n**
FILLER	Character	10 bytes	**A10**

The assignment of @fields to the unpack( ) function leaves us with 8 elements which represent the data (converted) for each record we read. There are a number of template conversions. Below is a list of the possible types of conversions.

A	An ascii string, will be space padded
a	An ascii string, will be null padded
c	A signed char value
C	An unsigned char value
s	A signed short value
S	An unsigned short value
i	A signed integer value
I	An unsigned integer value
l	A signed long value
L	An unsigned long value

n	A short (big endian)
N	A long in (big endian)
f	A single-precision float in the native format
d	A double-precision float in the native format
p	A pointer to a string
v	A short (little-endian)
V	A long in (little-endian)
x	A null byte
X	Back up a byte
@	Null fill to absolute position
u	A uuencoded string
b	A bit string (ascending bit order, like vec())
B	A bit string (descending bit order)
h	A hex string (low nibble first)
H	A hex string (high nibble first)

I had to use 'n' and 'N' instead of 'i', 'I', 'l', and 'L'. I am not sure at the time of this writing why that is (sorry). Is just is. Once I have the decoded fields for each record I want to print them. I do this by creating and calling a subroutine named print_records.

```
  print_recs();
```

It is there that I also print out the field name.

```
sub print_recs {

  for (my $i=0; $i<@fields; $i++)  {

     print "$field_names[$i]:  $fields[$i] \n" if $i < 7;

     print "\n" if $i == 6;

     exit if ++$ii > 23;

  }

}
```

I recommend that if you want to read binary files that you…

- Get a good hex editor
- Learn/Relearn how to read hex and binary and do conversion
- Be prepared to experiment

A more robust example

The following is a fairly lengthy program. I know that it is getting away from the toy code concept, but it does offer a nice insight into how one might go about abstracting a program for more general use. It is not complete by any means, but I decided I would go over it anyway.

READBINARY.pl / CHARGE.htm / CHARGE.dat

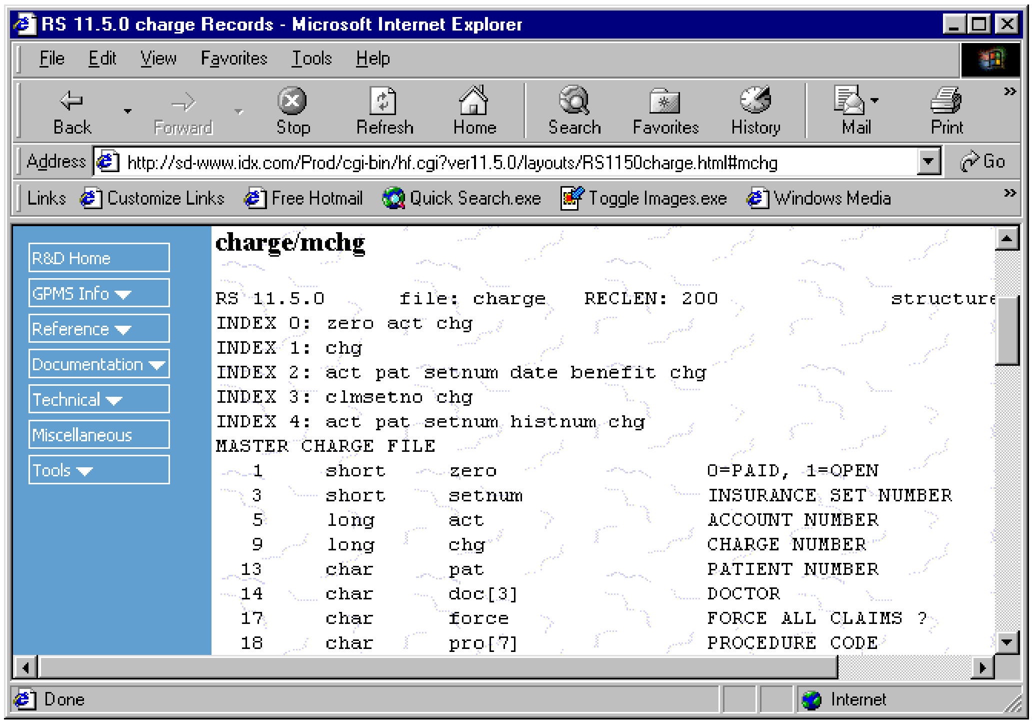
Why do this?

This is a good question. I was recently asked by our systems division to come out and do a class. They wanted to learn Perl for a myriad of reasons, one of which was to read ISAM files, and to write ISAM files.

I told them to send me a binary file and the schema for it. I also told them to pick a small one. I probably should have been more specific and said "small in respect to the number of fields" not the size of the file. They send me a file that was 200 bytes and had 59 fields. I must admit it is nice to get a dose of reality once in a while. I thought to myself that there was no way that I am going to type in translations for 59 fields. So I did so for the first 6 fields and gaffed off the rest.

I realized that even though I am Mr. Toy code I should put together an example that built the translation dynamically. After all this was real world. To boot these folks are systems administrators that are not programmers, though they are probably pretty adept at hacking code. I decided to put together a more practical and reusable example that addressed the "real world" problem.

The documentation schema for their ISAM files is accessible through the web. The following is a screenshot example for a file named charge.dat.


RS 11.5.0 charge Records - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search Favorites History Mail Print
Address http://sd-www.idx.com/Prod/cgi-bin/hf.cgi?ver11.5.0/layouts/RS1150charge.html#mchg
Links Customize Links Free Hotmail Quick Search.exe Toggle Images.exe Windows Media
R&D Home
GPMS Info
Reference
Documentation
Technical
Miscellaneous
Tools
Done
Internet

charge/mchg

```
RS 11.5.0       file: charge   RECLEN: 200               structure
INDEX 0: zero act chg
INDEX 1: chg
INDEX 2: act pat setnum date benefit chg
INDEX 3: clmsetno chg
INDEX 4: act pat setnum histnum chg
MASTER CHARGE FILE
   1      short     zero                    0=PAID, 1=OPEN
   3      short     setnum                  INSURANCE SET NUMBER
   5      long      act                     ACCOUNT NUMBER
   9      long      chg                     CHARGE NUMBER
  13      char      pat                     PATIENT NUMBER
  14      char      doc[3]                  DOCTOR
  17      char      force                   FORCE ALL CLAIMS ?
  18      char      pro[7]                  PROCEDURE CODE
```

The first goal was to read this file and build a translation template. I saw no point in hand coding translations. For one, that process is way too laborious and exceptionally prone to error by even the most diligent individual. The html file was pretty easy to read as much of what we needed was stock text wrapped inside a <pre> tag. The following is the subroutine that accomplishes this.

Code walk through

```
sub parse_schema_file {

while (<FS>) {

   $_ =~ /^ *[0-9]{1,3}( *uns *| +)(short|long|char) +([a-z0-
9]+\[*([0-9]+)*\]*) +/i;

   if ($2) {

     if ($2 eq "short") {

      $short++;

      $unpack_template = $unpack_template . $twobytes;

      $field_names[$fni++] = $3;

     }
```

```
    if ($2 eq "long") {
     $long++;
     $unpack_template = $unpack_template . $fourbytes;
     $field_names[$fni++] = $3;
    }
   if ($2 eq "char") {
     if ($4) {
       $iChar++;
       $totchars += $4;
       $unpack_template = $unpack_template . "A" . $4;
       $field_names[$fni++] = $3;
       $chartotal = $chartotal + $4;
     }
     else {
       $iChar++;
       $totchars += 1;
       $unpack_template = $unpack_template . "A" . 1;
       $field_names[$fni++] = $3;
       $chartotal = $chartotal + 1;
     }
    }
   }
} #end loop
} #end sub
```

While this code looks a bit nasty at first glance it is really not all that complicated. It would first be appropriate to explain the high level goals of this subroutine. Our first goal is to dynamically build the "template" for the unpack function. In order to do this we need to parse the lines in the html page that contain this schema. Here is a short sampling of the web page code we are interested in.

```
  1     short     zero                     0=PAID, 1=OPEN
  3     short     setnum                   INSURANCE SET NUMBER
  5     long      act                      ACCOUNT NUMBER
  9     long      chg                      CHARGE NUMBER
 13     char      pat                      PATIENT NUMBER
 14     char      doc[3]                   DOCTOR
 17     char      force                    FORCE ALL CLAIMS ?
```

The second column contains the data type. This column is quite important to us as it is needed to determine the translation we wish to apply. This file only has three data types, so that is all I coded for. You can see that I am extracting this column in my regular expression to $2.

```
   $_ =~ /^ *[0-9]{1,3}( *uns *| +)(short|long|char) …etc…
```

To make this more generalized and robust I would probably opt to code all the possible data type for this file type into a scalar and then substitute the hard coded values for the scalar. You can see that I am using (refer to the subroutine code above) $2 as a test for my nested conditionals. The first test…

```
   if ($2)
```

…is used so that we only process the actual schema lines of code. If $_ is actually a line of schema code in our current loop iteration then $2 will be populated and hold the data type. If not, then it will be undefined. Assuming $2 is defined, the inner conditionals are testing for the data type. For example:

```
    if ($2 eq "short")

    if ($2 eq "long")

    if ($2 eq "char")
```

It is within this conditional that we build the translation template. It is also here that I am keeping track of other vital information, like how many total bytes are record is. Processing can be a bit different dependent on the data type. For example, with the char data type we have a precision that can be expressed as *1 to n* characters. Since this value is variable we most certainly need to treat this as a special case. Lets example the inner conditional code.

```
     if ($2 eq "short") {

      $short++;

      $unpack_template = $unpack_template . $twobytes;

      $field_names[$fni++] = $3;
```

$short is a scalar I am using to keep track of how many short integers the record definition contains.. In the second line in the conditional I am updating the template by concatenating $unpack_template with the value of $twobytes which is a quasi-constant set to the value 'n' (a short in big endian order). In the last line I am adding to the array $field_names the name of the field. This is possible through assignment of the value we extracted into $3 of our regular expression.

```
([a-z0-9]+\[*([0-9]+)*\]*)
```

This is a nasty little bugger that I promise to explain in a moment. For now just know that this captures the third column of the a schema line.

The long conditional works pretty much the same as short. The only difference is that we are keeping a running total of how many long integers we have in the record definition by incrementing the scalar $long.

The char conditional is a bit trickier for us as the precision of the char can vary. For example, consider the following two lines.

```
  13     char      pat                     PATIENT NUMBER
  14     char      doc[3]                  DOCTOR
```

'pat' is a single character, whereas 'doc' is a char with a precision of three. I promised to explain the regular expression fragment…

```
([a-z0-9]+\[*([0-9]+)*\]*)
```

This is a tricky little piece of code. What is captured in $3 is the entire third column. The third column is our field name which will include the precision if the field is a char. We need to know the char precision for building the template. But, there will be no precision indicator if the char is a single character or another data type. This is quite problematic for us. The solution is to nest $4 parenthesis memory within $3 to account for this condition.

```
\[*([0-9]+)*\]*
```

Fortunately, we know the precision value will be enclosed inside a set of brackets so we can look for these. Since '[' and ']' will only exist for chars greater than 1 we need to use the * quantifier. Inside the parenthesis we will be looking to capture one of more numbers.

```
[0-9]+
```

But, these numbers will only exist for char data types greater than one so we need to use the * quantifier here as well.

```
([0-9]+)*
```

Now we can take a look at the inner conditionals that deal with char. Assuming $2 hold 'char'…

```
   if ($2 eq "char") {
```

…we can further figure out all that we need to construct the template.

```
     if ($4) {

       $iChar++;

       $totchars += $4;

       $unpack_template = $unpack_template . "A" . $4;

       $field_names[$fni++] = $3;

       $chartotal = $chartotal + $4;

     }
```

Our first test is against $4. If there is a precision for char greater than one we will have captured it to $4, hence it will be defined. $iChar is used to keep a running total of how many char data types are in the record definition. We are also keeping a running total of all the char bytes in our record, and do so by incrementing $totchars with the value we captured in $4.

```
$totchars += $4;
```

Next we update the template.

```
$unpack_template = $unpack_template . "A" . $4;
```

And, we capture the field name and increment $chartotal.

If we have a char line, but it does not have a precision we can assume that its value is one, and this is what we do in the else clause.

```
    else {

       $iChar++;

       $totchars += 1;

       $unpack_template = $unpack_template . "A" . 1;

       $field_names[$fni++] = $3;

       $chartotal = $chartotal + 1;

     }
```

Before we read the binary file I want to point out a few other aspects of the program. The program captures vital statisitcs regarding the schema file. For one, I calculate the total record length so that it can be compared to the html file.

```
$RECSZ = ($short * 2) + ($long * 4) + $chartotal;
```

I also keep trach of the total number of fields.

```
$totalfields = $short + $long + $iChar;
```

You can use the subroutine 'schema_statistics' to get a list of the vital statistics.

```
sub schema_statistics

{

 print "\n";

 print "   shorts:   $short \n";

 print "    longs:   $long \n";

 print "    chars:   $iChar \n";

 print "tot chars:   $totchars \n";

 print " template:   $unpack_template \n";

 print "tot field:   $totalfields\n";

 print " rec size:   $RECSZ\n";

}
```

Now back to the main road. Once we have constructed the translation template we can read the binary file. The subroutine ‘parse_binary_file’ is used to accomplish this. This subroutine also provides an example of a nested subroutine.

```
sub parse_binary_file {

 open (FB, $fil_binary) or die $!;

 binmode FB;

 while (<FB>) {

  read (FB, $record, $RECSZ);

  @fields = unpack($unpack_template,$record) ;

  print_recs();

 }

 my $ii=0;

 sub print_recs #example of nested sub  {

  for (my $i=0; $i<@fields; $i++)
  {
   print "$field_names[$i]: $fields[$i] \n" if $i <= $totalfields;
     print "\n" if $i == ($totalfields - 1);
  } }
}
```

We are using the read( ) function to read the file. read( ) takes three arguments. One, the file handle; two, a chuck of data; and three, the size of the chunk we want to read. Since we already know our record size is 200 bytes this establishes that until we hit eof on our filehande FB for our example that we will capture 200 byte chucks to $record.

```
read (FB, $record, $RECSZ);
```

To get to field level granularity for each record we use the unpack( ) function and assign the results to an array.

```
@fields = unpack($unpack_template,$record)
```

Remember, $unpack_template is our translation template that we constructed programmatically though the subroutine parse_schema_file. This template is applied to the 200 bytes we have in $record. The results end up in @fields. Since we have 59 fields in our example this will be the size of @array (though it is not important or necessary to know this).

For each record we then call the nested subroutine print_recs. I have opted to direct output to the console. In reality we would likely want to capture this output to a file.

Running the program

While we have a lot more work to further abstract this program we can still use it to deal with our scenario. This program will read any ISAM binary file by first reading our html documentation to create the template. Given this, you need to supply two arguments when running the program.

```
perl -w getinfo.pl <documentation_file> <binary_file>
```

Example:

```
perl -w getinfo.pl charge.htm charge.dat
```

Full code listing

Since you may not have access to the source code file I have decided to include the full program here.

READBINARY.pl

```
use strict;
my $fil_schema = $ARGV[0];
my $fil_binary = $ARGV[1];
open (FS, $fil_schema) or die $!;

# the next two are coded this way so they can be handled by command
# lines args. Or better yet, the 3rd arg should be file type, and then
# we could have a sub that set these values
my $twobytes = "n";
my $fourbytes = "N";

my $short=0;
my $long=0;
my @char;
my $iChar=0;
my $totchars=0;
my $totalfields=0;
my $unpack_template;
my $record;
my $RECSZ=0;
my @fields;
my @field_names;
my $fni=0; # field_names index
my $chartotal=0;



parse_schema_file();
```

```
$totalfields = $short + $long + $iChar;
$RECSZ = ($short * 2) + ($long * 4) + $chartotal;
parse_binary_file();
schema_statistics();


# This sub is specific to the schema format published as published on the
# web for our ISAM files. I do not think that it would be that hard to
# abstract this sub to accept a user defined template
sub parse_schema_file
{
while (<FS>)
{
   $_ =~ /^ *[0-9]{1,3}( *uns *| +)(short|long|char) +([a-z0-9]+\[*([0-
9]+)*\]*) +/i;
   if ($2)
   {
     if ($2 eq "short")
     {
      $short++;
      $unpack_template = $unpack_template . $twobytes;
      $field_names[$fni++] = $3;
     }


    if ($2 eq "long")
    {
     $long++;
     $unpack_template = $unpack_template . $fourbytes;
     $field_names[$fni++] = $3;
    }
   if ($2 eq "char")
   {
     if ($4)
     {
       $iChar++;
       $totchars += $4;
       $unpack_template = $unpack_template . "A" . $4;
```

```
        $field_names[$fni++] = $3;
        $chartotal = $chartotal + $4;
      }
      else
      {
        $iChar++;
        $totchars += 1;
        $unpack_template = $unpack_template . "A" . 1;
        $field_names[$fni++] = $3;
        $chartotal = $chartotal + 1;
      }
     }
    }

} #end loop
} #end sub

sub parse_binary_file
{
 open (FB, $fil_binary) or die $!;
# binmode FB;
 while (<FB>)
 {
  read (FB, $record, $RECSZ);
  @fields = unpack($unpack_template,$record) ;
  print_recs();
 }

 my $ii=0;
 sub print_recs #example of nested sub
 {
  for (my $i=0; $i<@fields; $i++)
  {
     print "$field_names[$i]: $fields[$i] \n" if $i <= $totalfields;
     print "\n" if $i == ($totalfields - 1);
```

```
   }
  }
}


sub schema_statistics
{
 print "\n";
 print "   shorts:   $short \n";
 print "    longs:   $long \n";
 print "    chars:   $iChar \n";
 print "tot chars:   $totchars \n";
 print " template:   $unpack_template \n";
 print "tot field:   $totalfields\n";
 print " rec size:   $RECSZ\n";
}
```