Programming – What does an Engineer need?

- Occasionally, an Engineer will have to write a program!
  - Not all problems can be solved by spreadsheets
- What types of problems might require programming?
  - Generate complex data streams as input to another problem
  - Parse/collect information out of large data files
  - Write a program that runs other programs in a regression test
  - Convert data files from one format to another
- Many programs are throw away code – use once to complete a task, then forget about it.
- Usually an Engineer is under time pressure
  - Need to become very familiar with a 'favorite' programming language, and use it enough to become time efficient.

Some Assumptions

- Programming is not the main task in your engineering workday
- Most programs you write are small, throw away programs
  - less than 100 lines
  - Use a program one or twice, then forget about it
- Computer environment is either Unix-based or Windows based
- Many work environments use both Windows and Unix
  - Windows for productivity tools (Spreadsheets, Word processing, Powerpoint)
  - UNIX workstations/servers for compute intensive jobs

Desirable Features of a Programming Language for ‘throw-away’ code

- Powerful – get a lot done with a little code
- Flexible – be able to do many different types of tasks
  - GUIs, string processing, program control, number crunching, etc.
- Well documented
- Large user base so external libraries, examples readily available
- Portable – be able to run on different systems
  - carry your favorite code with you when you change jobs
- Readily available (“free” is the best!)

Compiled Programming Languages

- C, C++, Fortran are traditional compiled languages
- Pros
  - High performance code
  - Portable between systems
  - Free C, C++, Fortran compilers from the Free Software Foundation
- Cons
  - Usually have to write a lot of code to get even simple tasks done
  - Non-standard extension libraries which means you have to move your favorite library from system to system
  - Must compile source code first on target system before execution.
  - GUI interfaces are Operating System dependent
  - Support for ‘scripting’ (i.e, controlled execution of other programs) is minimal and Operating System dependent
- Best for large, complex tasks -- but may not be best choice for simple tasks (i.e, < 100 lines of code)

UNIX Shell Scripting Languages

- All UNIX shells (csh, bash, ksh, etc) support a scripting language
- Pros
  - Builtin to shell, always available for use
  - Ideal for scripting duties (control of other programs)
- Cons
  - Fairly primitive features, no powerful operators or data handling features
  - Very slow – at least 100x slower than compiled code
  - No GUI capabilities
  - Only useful for Unix applications

Visual Basic

- Visual Basic is the most common scripting/tool extension language on Windows platforms
- Pros
  - Integrated with Windows productivity tools (spreadsheets, etc) – can be used to extend their base capabilities
  - Very nice GUI building capabilities
  - Complex data types, powerful library functions
  - Has scripting capabilities
  - Decent performance (about 10x less than compiled C/C++)
- Cons
  - only works under Windows OS
  - development environment costs $$$
- Might be the best choice for throw-away code if you never touch a Unix system

Java

- Portable object-oriented programming language
- Pros
  - Powerful data structures, functions
  - Portable between Unix/Windows
  - Free development environment
  - Powerful GUI building
  - Useful for Web page enhancement
  - Decent performance (about 10x less than compiled C/C++)
- Cons
  - Limited scripting, string processing
  - Object-oriented programming model is possible overkill for simple throw-away programs

Perl

- Scripting language that combines the best of Unix shell languages plus powerful string handling and builtin functions
- Pros
  - Powerful data structures, functions
  - Portable between Unix/Windows
  - Free development environment
  - Decent performance (about 10x less than compiled C/C++)
  - Many public libraries available for tasks such as HTML processing and data base access
  - Extremely large user base – the scripting language of choice under Unix
- Cons
  - No GUI building capabilities
  - Code may be unreadable after you write it!
- In my opinion, best choice for throw away code development under Unix environment, and perhaps a combined Windows/Unix environment.

Others

- Tcl/Tk
  - Portable scripting (TCL) + GUI development (TK) environment
  - A better choice than Perl if you need both sophisticated scripting and GUI development in the same programming language
  - Free implementations for both Windows and Unix
- Python
  - Best described as an object oriented scripting language
  - Has powerful GUI building features
  - Free implementations for both Windows and Unix
  - Better choice than Perl if you need sophisticated scripting + GUI development, and you like the object-oriented programming model

Where to learn more

- http://www.activestate.com
  - Free implementations of Perl, TCL/TK, Python for Win32
- Java – http://java.sun.com
- Visual Basic
  - Limited form of VB comes with Excel
  - Start Excel, go to Tools→Macro →Visual Basic Editor
  - Access help function under Visual Basic Editor
  - Full version requires purchase of Visual Basic Studio (http://msdn.microsoft.com/vbasic/default.asp)
- UWIN comes with Gnu compilers for C, C++, Fortran

Will look at Perl in ECE 3732

- Why?
  - Good example of a scripting language
  - Portable between Windows/Unix
  - Dr. Reese's favorite language for throw-away code
- Will you ever use Perl again?
  - Maybe, Maybe not – depends on career path, job function, and if you see Perl as being useful for throw away tasks
- Will you ever have to write a program again after ECE 3732?
  - If you are an engineer, you WILL have to write a program occasionally
  - What language you use will often be entirely your choice – you will have some task to perform and writing a program will help you do that task more efficiently.
  - Pick a language for throw away programming tasks, and use it enough to become time efficient!!!!

Perl Distributions

- You have two Perl distributions on your CDROM
- UWIN Perl
  - Installation file is Uwin/UWIN Dist/uwin_perl.win32.i386.exe
  - After install, run from ksh window, executable is /usr/bin/perl
- ActiveState Perl
  - Installation file is Uwin/actperl/ActivePerl-5.6.1.628-MSWin32-x86-multi-thread.msi
  - After install, run from MSDOS prompt, binary will be in C:\Perl\bin\Perl.exe
- Install both on your systems, Active Perl primarily for documentation reasons

Perl Documentation

- Active Perl installation has best documentation
  - C:\Perl\html\index.html
- General Perl sites
  - http://www.perl.com , http://www.perl.org, http://www.activeperl.com
- Definitive textbook for Perl
  Programming Perl 3rd edition, L. Wall, T. Christiansen, R. Schwarz
  Published by O'Reilly, $35 on Amazon.com
- Online Perl tutorials
  - http://www.comp.leeds.ac.uk/Perl/start.html

    (all of these initial notes are copied from the above tutorial)

A Perl Script

The file below is a perl script that prints out "Hello World".

File: hworld.pl

```
print "Hello World!\n";
```

To execute this script (example shown from *ksh*)

```
$ perl hworld.pl   ←
Hello World!
$
```

Pass script name as argument to perl

Can also use command line arguments such as:

```
$ perl –w hworld.pl
```

The –w argument is nice, provides some warnings about common mistakes in Perl.

Making hworld.pl executable

Can make the *hworld.pl* script directly executable by adding the following line:

```
#!/usr/bin/perl   ←
print "Hello World!\n";
```

First line is pathname to perl executable

Must also give file execute permission:

```
$ chmod +x hworld.pl
```

Now can directly execute the file:

```
$ hworld.pl
Hello World!
$
```

Scalar Variables

The most basic kind of variable in Perl is the *scalar variable*. Scalar variables hold both strings and numbers, and are remarkable in that strings and numbers are completely interchangable.
For example, the statement

```
$priority = 9;
```

sets the scalar variable *$priority* to 9, but you can also assign a string to exactly the same variable:

```
$priority = 'high';
```

Perl also accepts numbers as strings, like this:

```
$priority = '9'; $default = '0009';
```

and can still cope with arithmetic and other operations quite happily. In general variable names consists of numbers, letters and underscores, but they should not start with a number and the variable **$_** is special, as we'll see later. Also, Perl is case sensitive, so $a and $A are different.

Operators

Perl uses all the usual C arithmetic operators:

```
$a = 1 + 2;          # Add 1 and 2 and store in $a
$a = 3 - 4;          # Subtract 4 from 3 and store in $a
$a = 5 * 6;          # Multiply 5 and 6
$a = 7 / 8;          # Divide 7 by 8 to give 0.875
$a = 9 ** 10;        # Nine to the power of 10
$a = 5 % 2;          # Remainder of 5 divided by 2
++$a;                # Increment $a and then return it
$a++;                # Return $a and then increment it
--$a;                # Decrement $a and then return it
$a--;                #Return $a and then decrement it
```

and for strings Perl has the following among others:

```
$a = $b . $c;        # Concatenate $b and $c
$a = $b x $c;        # $b repeated $c times
```

Assigning Perl Values

To assign values Perl includes

```
$a = $b;      # Assign $b to $a
$a += $b;     # Add $b to $a
$a -= $b;     # Subtract $b from $a
$a .= $b;     # Append $b onto $a ($a, $b contain strings)
```

Note that when Perl assigns a value with **$a = $b** it makes a copy of $b and then assigns that to $a. Therefore the next time you change $b it will not alter $a.

Other operators can be found on the **perlop** manual page.

In the ActiveWin html documentation, look under "**Core Perl Docs**", and look at *perlop*

Interpolation of Variable Names

The following code prints *apples and pears* using concatenation (note the use of the '.' (period) as the string concatenation operator).

```
$a = 'apples';
$b = 'pears';
print $a . ' and ' . $b;
```

It would be nicer to include only one string in the final print statement, but the line

```
print '$a and $b';
```

prints literally *$a and $b* which isn't very helpful. Instead we can use double quotes in place of the single quotes:

```
print "$a and $b";
```

The double quotes force *interpolation* of any codes, including interpreting variables. Other codes that are interpolated include special characters such as newline and tab. The code **\n** is a newline and **\t** is a tab.

Array Variables

A slightly more interesting kind of variable is the *array variable* which is a list of scalars (ie numbers and strings). Array variables have the same format as scalar variables except that they are prefixed by an @ symbol. The statement

```
@food = ("apples", "pears", "eels");
@music = ("whistle", "flute");
```

assigns a three element list to the array variable @food and a two element list to the array variable @music.

The array is accessed by using indices starting from 0, and square brackets are used to specify the index. The expression

```
$food[2]
```

returns *eels*. Notice that the **@** has changed to a **$** because *eels* is a scalar.

Array Assignments

As in all of Perl, the same expression in a different context can produce a different result. The first assignment below explodes the @music variable so that it is equivalent to the second assignment.

```
@moremusic = ("organ", @music, "harp");
@moremusic = ("organ", "whistle", "flute", "harp");
```

This should suggest a way of adding elements to an array. A neater way of adding elements is to use the statement

```
push(@food, "eggs");
```

which pushes *eggs* onto the end of the array @food. To push two or more items onto the array use one of the following forms:

```
push(@food, "eggs", "lard");
push(@food, ("eggs", "lard"));
push(@food, @morefood);
```

The **push** function returns the length of the new list.

More List Manipulation

```
@food = ("eels", "lard", "bread" );
```

It is possible to assign an array to a scalar variable. Examples are:

```
$f = @food;          # $f = length of @food, here = 3.
$g = $#food;         # $g = last index of @food, here = 2
```

The assignment:

```
$f = “@food”;
```

turns the value of $f into a string that are the members of @food with a space between each element.
The assignment below assigns $g the last item in the list @food:

```
$g = $food[$#food];    # $g = $food[2] = “bread”;
```

To remove the last item from a list and return it use the **pop** function. Note that after this assignment, $g = “bread”, and @food only has 2 elements left.

```
$g = pop(@food);
```

File Handling

Here is a simple perl program which does the same as the UNIX **cat** command on a certain file.

```
#!/usr/bin/perl
#
# Program to open the password file, read it in,
# print it, and close it again.

$file = '/etc/passwd';      # Name the file
open(INFO, $file);          # Open the file
@lines = <INFO>;            # Read it into an array
close(INFO);                # Close the file
print @lines;               # Print the array
```

The **open** function opens a file for input (i.e. for reading). The first parameter is the *filehandle* which allows Perl to refer to the file in future. The second parameter is an expression denoting the filename. A *filehandle* name can be any valid Perl symbol name.

The **close** function tells Perl to finish with that file.

More File Handling

The **open** statement can also specify a file for output and for appending as well as for input. To do this, prefix the filename with a > for output and a >> for appending:

```
open(INFO, $file);          # Open for input
open(INFO, ">$file");       # Open for output
open(INFO, ">>$file");      # Open for appending
open(INFO, "<$file");       # Also open for input
```

To print a string to the file with the INFO filehandle use:

```
print INFO "This line goes to the file.\n";
```

To open the standard input (usually the keyboard) and standard output (usually the screen) respectively do:

```
open(INFO, '-');     # Open standard input
open(INFO, '>-');    # Open standard output
```

foreach Control Structure

To go through each line of an array or other list-like structure (such as lines in a file) Perl uses the foreach structure. This has the form:

```
foreach $morsel (@food)     # Visit each item in turn
                            # and call it $morsel
{
       print "$morsel\n";   # Print the item
       print "Yum yum\n";   # That was nice
}
```

The actions to be performed each time are enclosed in a block of curly braces. The first time through the block $morsel is assigned the value of the first item in the array @food. Next time it is assigned the value of the second item, and so until the end. If @food is empty to start with then the block of statements is never executed.

Testing

The next few structures rely on a test being true or false. In Perl any non-zero number and non-empty string is counted as true. The number zero, zero by itself in a string, and the empty string are counted as false. Here are some tests on numbers and strings.

```
a == $b             # Is $a numerically equal to $b?
                    # Beware: Don't use the = operator.
$a != $b            # Is $a numerically unequal to $b?
$a eq $b            # Is $a string-equal to $b?
$a ne $b            # Is $a string-unequal to $b?
```

You can also use logical and, or and not:

```
($a && $b)          # Is $a and $b true?
($a || $b)          # Is either $a or $b true?
!($a)               # is $a false?
```

for Control Structure

Perl has a **for** structure that mimics that of C. It has the form

```
for (initialise; test; inc)
{
        first_action;
       second_action;
       etc
}
```

First of all the statement *initialise* is executed. Then while *test* is true the block of actions is executed. After each time the block is executed *inc* takes place. Here is an example for loop to print out the numbers 0 to 9.

```
for ($i = 0; $i < 10; $i++)  # Start with $i = 0
                             # Do it while $i < 10
                             # Increment $i before repeating
 {
      print "$i\n";
 }
```

while

Here is a program that reads some input from the keyboard and won't continue until it gets the correct password

```
#!/usr/bin/perl
print "Password? ";         # Ask for input
$a = <STDIN>;               # Get input
chop $a;                    # Remove the newline at end
while ($a ne "fred")        # While input is wrong...
{
       print "sorry. Again? ";    # Ask again
       $a = <STDIN>;              # Get input again
       chop $a;                   # Chop off newline again
}
```

When the password is entered $a is given that value including the newline character at the end. The **chop** function removes the last character of a string which in this case is the newline. The “ne” function stands for “not equal” and is a string comparison function (“eq” is the string equality test).

do - until

Here is a version of the previous program that uses a *do – until* control structure:

```
#!/usr/local/bin/perl
do
{
       "Password? "; # Ask for input
       $a = <STDIN>; # Get input chop
       $a;          # Chop off newline
}
while ($a ne "fred") # Redo while wrong input
```

if, elsif statements

Of course Perl also allows if/then/else statements. These are of the following form:

```
if ($a) {
       print "The string is not empty\n";
} else {
       print "The string is empty\n";
}
```

elsif is a way of chaining multiple if-else statements (note the spelling on *elsif*):

```
if (!$a){     # The ! is the not operator
       print "The string is empty\n";
} elsif (length($a) == 1) { # If above fails, try this
       print "The string has one character\n";
} elsif (length($a) == 2) { # If that fails, try this
       print "The string has two characters\n";
} else {      # Now, everything has failed
       print "The string has lots of characters\n";
}
```

Regular Expressions

A regular expression is contained in slashes, and matching occurs with the =~ operator. The following expression is true if the string *the* appears in variable $sentence.

```
$sentence =~ /the/
```

The RE is case sensitive, so if

```
$sentence = "The quick brown fox";
```

then the above match will be false. The operator **!~** is used for spotting a non-match. In the above example

```
$sentence !~ /the/
```

is true because the string *the* does not appear in $sentence.

The *$_* special variable

We could use a conditional as

```
if ($sentence =~ /under/) {
       print "Sentence has 'under' in it\n";
}
```

which would print out a message if we had either of the following

```
$sentence = "Up and under";
$sentence = "Best winkles in Sunderland";
```

But it's often much easier if we assign the sentence to the special variable **$_** which is of course a scalar. If we do this then we can avoid using the match and non-match operators and the above can be written simply as

```
$_ = $sentence;
if (/under/) {
       print "Sentence has 'under' in it\n";
}
```

Many Perl functions return values in $_, so explicit assignment to $_ is usually unnecessary.

More on REs

In an RE there are plenty of special characters, and it is these that both give them their power and make them appear very complicated. It's best to build up your use of REs slowly; their creation can be something of an art form.

Here are some special RE characters and their meaning

```
.      # Any single character except a newline
^      # The beginning of the line or string
$      # The end of the line or string
*      # Zero or more of the last character
+      # One or more of the last character
?      # Zero or one of the last character
```

These special characters can be embedded in REs to add power to the search.

Example REs

here are some example matches – there is a lot more to REs but we will stop here.

```
/.e/          # t followed by anthing followed by e
              # This will match the
              # tre
              # tle
              # but not te
              # tale
/^f/          # f at the beginning of a line
/^ftp/        # ftp at the beginning of a line
/e$/          # e at the end of a line
/tle$/        # tle at the end of a line
/und*/        # un followed by zero or more d characters
              # This will match un
              # und
              # undd
              # unddd (etc)
/.*/          # Any string without a newline. This is because
              # the . matches anything except a newline and
              # the * means zero or more of these.
/^$/          # A line with nothing in it.
```

Substitution

To replace an occurrence of *london* by *London* in the string $sentence we use the expression

```
$sentence =~ s/london/London/
```

and to do the same thing with the **$_** variable just

```
s/london/London/
```

This example only replaces the first occurrence of the string, and it may be that there will be more than one such string we want to replace. To make a global substitution the last slash is followed by a **g** as follows (substitution occurs in the $_ variable):

```
s/london/London/g
```

The *i* option can used to ignore case:

```
s/london/London/gi
```

Translation

The **tr** function allows character-by-character translation. The following expression replaces each *a* with *e*, each *b* with *d*, and each *c* with *f* in the variable $sentence. The expression returns the number of substitutions made.

```
$sentence =~ tr/abc/edf/
```

The following statement converts **$_** to upper case.

```
tr/a-z/A-Z/;
```

Because the translated result returns in $_, can have multiple *tr* statements follow one another.

```
tr/;//g;
tr/://g;
tr/#//g;
```

The above translations replace ';', ':', '#' with the null character, removing them from the string stored in $_.

split Function

A very useful function in Perl is **split**, which splits up a string and places it into an array. The function uses a regular expression and as usual works on the **$_** variable unless otherwise specified.

The **split** function is used like this:

```
$info = "Caine:Michael:Actor:14, Leafy Drive";
@personnel = split(/:/, $info);
```

which has the same overall effect as

```
@personnel = ("Caine", "Michael", "Actor", "14, Leafy Drive");
```

If we have the information stored in the **$_** variable then we can just use this instead

```
@personnel = split(/:/);
```

More on *split*

A word can be split into characters, a sentence split into words and a paragraph split into sentences:

```
@chars = split(//, $word);
@words = split(/ /, $sentence);
@sentences = split(/\./, $paragraph);
```

Check the length of the returned array to see the string was actually split:

```
@words = split(/:/, "hello there");

if (@words == 1) {
   print "String did not contain a ':'\n";
} else {
   $i = @words;
   print "String was split into $i words.\n";
 }
```

Associative Arrays

Ordinary list arrays allow us to access their element by number. The first element of array @food is $food[0]. The second element is $food[1], and so on. But Perl also allows us to create arrays which are accessed by string. These are called *associative arrays*.

To define an associative array we use the usual parenthesis notation, but the array itself is prefixed by a % sign. Suppose we want to create an array of people and their ages. It would look like this:

```
%ages = ("Michael Caine", 39,
       "Dirty Den", 34,
       "Angie", 27,
       "Willy", "21 in dog years",
       "The Queen Mother", 108);
```

Note that entries in associative arrays occur in pairs.

Accessing Elements in Associative Arrays

Now we can find the age of people with the following expressions

```
$ages{"Michael Caine"};    # Returns 39
$ages{"Dirty Den"};        # Returns 34
$ages{"Angie"};            # Returns 27
$ages{"Willy"};            # Returns "21 in dog years"
$ages{"The Queen Mother"}; # Returns 108
```

An associative array can be converted back into a list array just by assigning it to a list array variable. A list array can be converted into an associative array by assigning it to an associative array variable. Ideally the list array will have an even number of elements:

```
@info = %ages;      # @info is a list array. It
                    # now has 10 elements
$info[5];           # Returns the value 27 from
                    # the list array @info
%moreages = @info;  # %moreages is an associative
                    # array. It is the same as %ages
```

Looping over Elements in Associative Arrays

Associative arrays do not have any order to their elements (they are just like hash tables) but is it possible to access all the elements in turn using the **keys** function and the **values** function:

```
foreach $person (keys %ages) {
	print "I know the age of $person\n";
}

foreach $age (values %ages) {
	print "Somebody is $age\n";
}
```

There is also a function **each** which returns a two element list of a key and its value. Every time **each** is called it returns another key/value pair:

```
while (($person, $age) = each(%ages)) {
	print "$person is $age\n";
}
```

Subroutines

Perl allows the user to define their own functions, called *subroutines*. They may be placed anywhere in your program but it's probably best to put them all at the beginning or all at the end. A subroutine has the form

```
sub mysubroutine {
	print "Not a very interesting routine\n";
	print "This does the same thing every time\n";
}
```

The above subroutine ignores any parameters passed to it. All of the following will work to call this subroutine. Notice that a subroutine is called with an **&** character in front of the name:

```
&mysubroutine;              # Call the subroutine
&mysubroutine($_);          # Call it with a parameter
&mysubroutine(1+2, $_);     # Call it with two parameters
```

Passing Parameters to Subroutines

When a subroutine is called any parameters are passed as a list in the special @_ list array variable. The following subroutine merely prints out the list that it was called with. It is followed by a couple of examples of its use.

```
sub printargs {
       print "@_\n";
}

&printargs("perly", "king"); # Example prints "perly king"
&printargs("frog", "and", "toad"); # Prints "frog and toad"
```

Just like any other list array the individual elements of @_ can be accessed with the square bracket notation:

```
sub printfirsttwo {
       print "Your first argument was @_[0]\n";
       print "and @_[1] was your second\n";
}
```

Returning Values from Subroutines

Result of a subroutine is always the last thing evaluated or the value explicitly passed in the ‘return’ statement. This subroutine returns the maximum of two input parameters. It is recommended that you always use ‘return’ for improved code readability:

```
sub maximum {
       if ($_[0] > $_[1]) {
              return($_[0]);
       } else {
              return($_[1]);
       }
}
$biggest = &maximum (37,24);  # $biggest gets the value 37
```

Local Variables

Local variables can be declared inside of a subroutine using the *my* function:

```
sub inside {
 my($a, $b);                        # Make local variables
 ($a, $b) = (@_[0], @_[1]);         # Assign values
 $a =~ s/ //g;                      # Strip spaces from
 $b =~ s/ //g;                      # local variables
 return ($a =~ /$b/ || $b =~ /$a/); # Is $b inside $a
                                        # or $a inside $b?
}

&inside("lemon", "dole money"); # returns true
```

Passing Arguments on the Command Line

The perl program uses arguments passed on the command line:

```
#!/usr/bin/perl
use English;
if (@ARGV < 2) {
    print "Usage: $PROGRAM_NAME arg1 arg2 \n";
    print "Echos arg1, arg2 to screen \n";
    exit(0);
}
print "Argument #1 is $ARGV[0]\n";
print "Argument #2 is $ARGV[1]\n";
```

Arguments passed on the command line are passed as a list in the special variable ARGV . The “use English;” statement enables longer, more informative variable names for Perl special variables. The $PROGRAM_NAME special variable is the name of the program specified on the command line (the shorter variable name is $0).

Example Runs

Below are some example runs of the program on the previous slide:

```
$ perl arg.pl
Usage: arg.pl arg1 arg2
Echos arg1, arg2 to screen
```

zero arguments passed to program, get usage string

```
$ perl arg.pl Hi!
Usage: arg.pl arg1 arg2
Echos arg1, arg2 to screen
```

only 1 argument passed, get usage string

```
$ perl arg.pl Hi! There!
Argument #1 is Hi!
Argument #2 is There!
```

correct number of arguments passed, program executes normally.

Including argument checking and ‘usage’ messages is good programming etiquette.