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.

BR Fall 2001

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

BR Fall 2001

2

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!)

BR Fall 2001

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)

BR Fall 2001

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

BR Fall 2001

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

BR Fall 2001

3

1

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

BR Fall 2001

7

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. BR Fall 2001 8

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

BR Fall 2001

- Where to learn more http://www.activestate.com - Free implementations of Perl, TCL/TK, Python for Win32 Java – <u>http://java.sun.com</u>
- 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

BR Fall 2001

10

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!!!!

BR Fall 2001

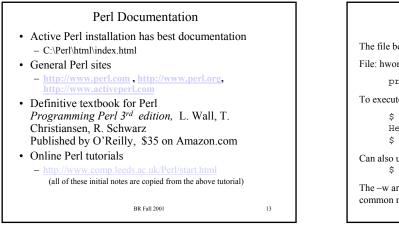
11

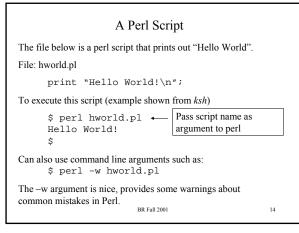
Perl Distributions

- · 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

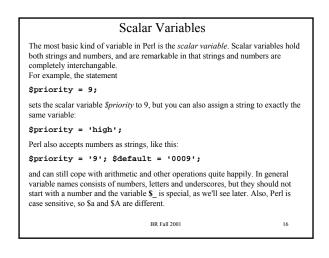
BR Fall 2001

12





Making hworld.pl executable			
Can make the <i>hworld.pl</i> script directly executable by adding t following line: #!/usr/bin/perl			
	print "Hello World!\n";		
	Must also give file execute permission:		
	<pre>\$ chmod +x hworld.pl</pre>		
	Now can directly execute the file:		
	\$ hworld.pl Hello World! \$		
	BR Fall 2001 15	5	



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;	<pre># Remainder of 5 divided by 2</pre>
++\$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	\$Ъ	and \$c
\$a = \$b x	\$c; #	\$b repeated	\$c	times

17

Assigning Perl Values

To assign values Perl includes

\$a = \$b;	# Assign \$b to \$a	
\$a += \$b;	# Add \$b to \$a	
Śa −= Śb;	# 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

BR Fall 2001

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 tt 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 <code>#food = ("apples", "pears", "eels"); music = ("whistle", "flute");</code> 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.

BR Fall 2001

20

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

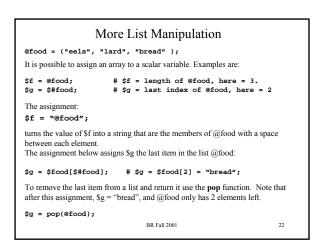
push(@food, "eggs");

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.

BR Fall 2001



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 the array print @lines; 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.

BR Fall 2001

23

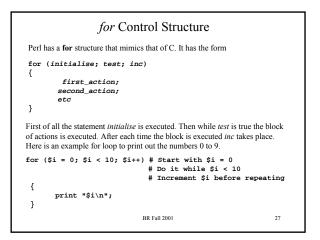
21

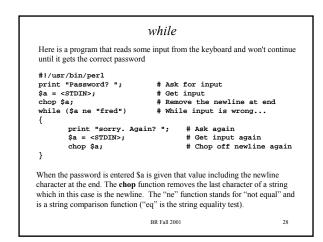
More File Handling The open statement can also specify a file for output and for appending as well as for input. To do this, prefix the filename with a > for output and a >> for appending: open(INFO, \$file); # Open for input open(INFO, ">\$file"); # Open for output open(INFO, ">>\$file"); # Open for appending open(INFO, "<\$file");</pre> # 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 BR Fall 2001 24

foreach Control Structure

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.

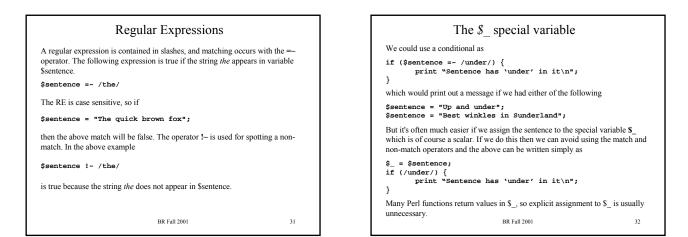
Testing The next few structures rely on a test being true or false. In Perl any non-zero number and non-empty string is counted as true. The number zero, zero by itself in a string, and the empty string are counted as false. Here are some tests on numbers and strings. \$a == \$b # Is \$a numerically 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? # Is \$a string-unequal to \$b? \$a ne \$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? BR Fall 2001 26





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 chop \$a; # Chop off newline } while (\$a ne "fred") # Redo while wrong input BR Fall 2001

<i>if, elsif</i> statements	
Of course Perl also allows if/then/else statements. These are of the followin form:	ıg
<pre>if (\$a) { print "The string is not empty\n"; } else { print "The string is empty\n"; }</pre>	
elsif is a way of chaining multiple if-else statements (note the spelling on el	lsif):
<pre>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"; }</pre>	
BR Fall 2001	30



nere are some of	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
/^£/	<pre># f at the beginning of a line</pre>
/^ftp/	<pre># ftp at the beginning of a line</pre>
/e\$/	# e at the end of a line
/tle\$/	<pre># tle at the end of a line</pre>
/und*/	
	# This will match un
	# und
	# undd
	<pre># unddd (etc)</pre>
/.*/	# Any string without a newline. This is becau
	# the . matches anything except a newline and
	# the * means zero or more of these.
/^\$/	# A line with nothing in it. BR Fall 2001 34

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

BR Fall 2001

35

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/

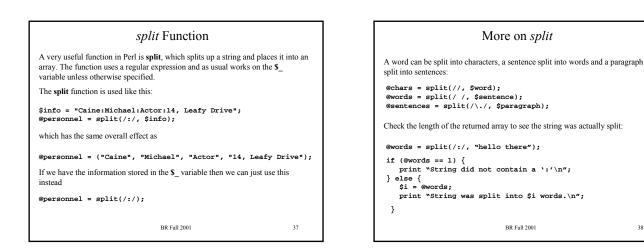
The following statement converts \$_ to upper case.

tr/a-z/A-Z/;

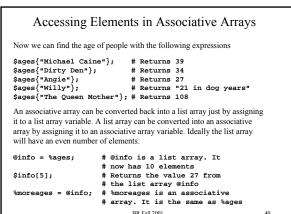
Because the translated result returns in \$_, can have multiple tr statements follow one another.

tr/;//g; tr/://g; tr/#//q;

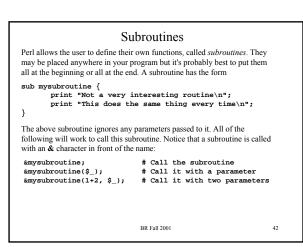
The above translations replace ';', ':', '#' with the null character, removing them from the string stored in \$. BR Fall 2001 36

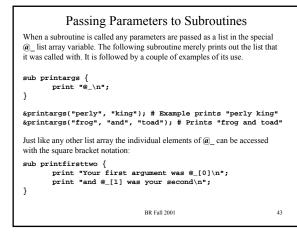


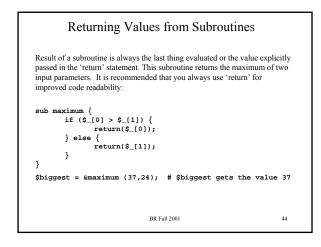
	r
Associative Arrays Ordinary list arrays allow us to access their element by number. The first element of array @food is \$food[0]. The second element is \$food[1], and so on. But Perl also allows us to create arrays which are accessed by string. These are called associative arrays.	
To define an associative 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:	
<pre>%ages = ("Michael Caine", 39, "Dirty Den", 34, "Angie", 27, "Willy", "21 in dog years",</pre>	
"The Queen Mother", 108); Note that entries in associative arrays occur in pairs.	
BR Fall 2001 39	

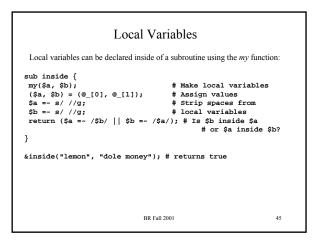


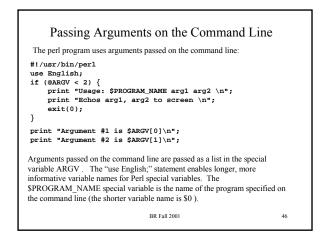
Looping over Elements in Associative Array: Associative arrays do not have any order to their elements (they are just like hash tables) but is it possible to access all the elements in turn using the key function and the values function: foreach \$person (keys %ages) { print "I know the age of \$person\n"; }	,
<pre>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)) {</pre>	
<pre>while ((\$person, \$age) = each(\$ages)) { print "\$person is \$age\n"; } BR Fall 2001</pre>	41











Example Below are some example runs of the prog	
<pre>\$ perl arg.pl Usage: arg.pl arg1 arg2 Echos arg1, arg2 to screen</pre>	zero arguments passed to program, get usage string
<pre>\$ perl arg.pl Hi! Usage: arg.pl arg1 arg2 Echos arg1, arg2 to screen</pre>	only 1 argument passed, get usage string
<pre>\$ perl arg.pl Hi! There! Argument #1 is Hi! Argument #2 is There!</pre>	correct number of arguments passed, program executes normally.
Including argument checking and 'usage' etiquette.	messages is good programming
BR Fall	2001 47