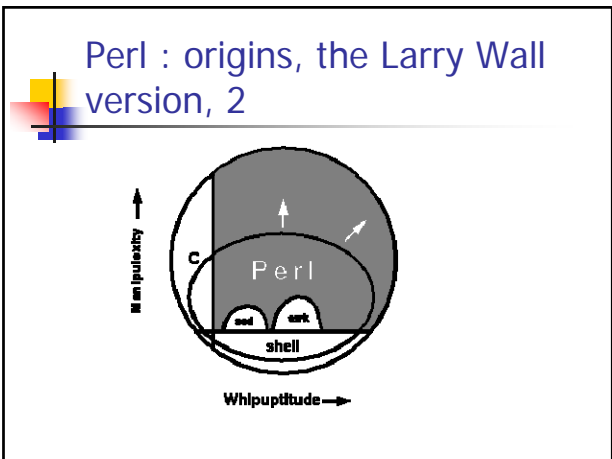
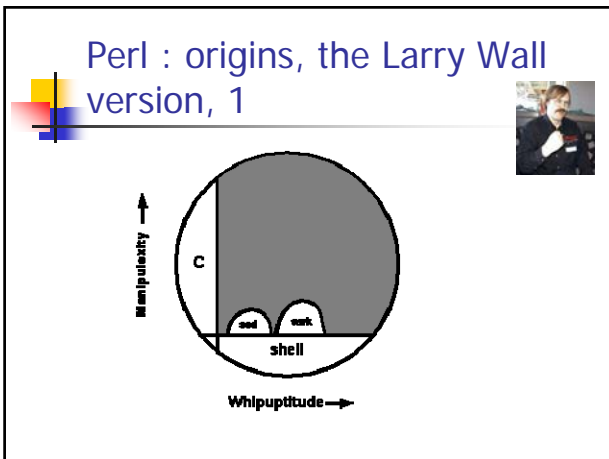


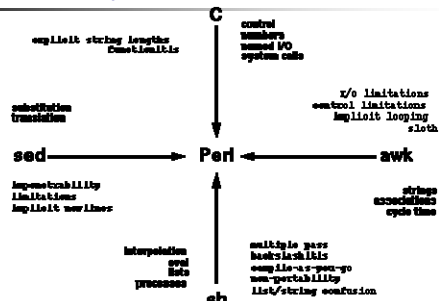
- ### Perl: *component objectives*
- To get you to be able to:
    - Write simple Perl programs for general text manipulation.
    - Use Perl libraries for database access.
    - Use Perl for CGI programming.

- ### Perl: origins
- Traditional Unix:
    - ed (vi) text file editor
      - Regular expression matcher, sophisticated global search and replacement operations.
    - sed
      - Script editor read a file with ed-style replacement commands, apply these to input file(s)
    - awk
      - Pattern scanning and replacement
        - Read input line by line,
        - Match lines against awk patterns
        - Perform specified processing
    - sh
      - File manipulations

- ### Perl: origins
- Practical Extraction and Report Language
    - Its original role:
      - *handle tasks similar to those where sed, and awk had been applied; use a slightly more familiar programming style.*



## Perl : origins, the Larry Wall version, 3



## Perl: interpreter

- Interpretive language
- Perl interpreter reads and pre-processes script, preparing code for execution.
- Perl code then runs, reading input(s) and performing operations
  - Mostly, but not entirely, text processing: finding and replacing strings, counting symbols etc.

## Perl: interpreter

- Perl interpreter written in C.
- Input and output operations in Perl script are implemented as calls to input and output functions written in C (functions that are part of the interpreter).
- I/O therefore as efficient as that of a C program.

## Perl: interpreter overheads?

- Most Perl programs tend to be I/O bound (*lots of text read and written, a few simple operations performed*).
- Since I/O is done in C, and since most of processing is I/O related, *Perl programs are **not** always significantly slower than hard coded C programs performing the same task.*

## Perl: files for interpreter

- You can
  - Have a file containing a Perl script,
  - Explicitly launch the perl interpreter with a command line argument identifying script:
 

```
$ perl myperlprog.pl
```
- **Usually**, take advantage of a Unix "feature" that makes Perl scripts appear directly executable
 

```
$ myperlprog.pl
```

## Perl: executable scripts

- Change directory entry for Perl file to mark it as executable (`chmod +x myperlprog.pl`)
- Have magic first line:
 

```
#!/usr/bin/perl
```
- Perl script now can just be "run".
 

```
$ myperlprog.pl
```

Depends on where perl installed; do "which perl"; my perl was in /share/bin/perl

## Perl: executable scripts

- Explanation:
  - Unix reads first two bytes of a supposed executable file and uses them to form a "magic number"
  - Different magic numbers define different mechanisms for dealing with rest of file.
  - `#!` Value means:
    - Use rest of this line as pathname to executable interpreter program
    - Start process running that interpreter
    - Feed rest of file to interpreter on its standard input

## Perl: the language ...

- Perl
  - Core defined by the interpreter
  - Extensions provided by libraries written in Perl
- Perl essentially "open source"
  - Lots of extensions contributed as Perl evolved, adding different features to the core.
  - Lots of libraries created.

## Perl: an eclectic mix ...

- Every contributor to Perl had his/her own favorite programming idioms, function libraries etc.
- ***ALL were incorporated.***
- Result:
  - There is always more than one way of doing anything in Perl.
  - The different ways appear totally unrelated.

## Hello Perl

## Perl: the inevitable ...

```
#!/usr/bin/perl
```

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

↑  
*Optional, it is a statement separator*

## Perl: introduction to language

## Perl language: 1

- *The usual:*
  - Flow control:
    - Sequence
    - Selection
    - Iteration
    - Function call
    - Object method call
  - **Mostly similar to C/C++ ; but naturally some syntactic differences**  
e.g. "if":
    - `if(condition) block`
    - rather than `if(condition)statement`

## Perl language: 2

- *The usual, continued:*
  - Literal constants
    - Numbers (12, 1.2, 1.2e3, 014 (octal), ...)
    - Strings "hello world", "hi mom"
    - Special character constants '\t', '\n', ...
  - Expressions
    - Operators defined for numeric and string types

## Perl language:3

- Differences!
  - Perl uses "type identifiers" along with variable names; type identifiers distinguish scalars (simple variables), arrays (lists), "hashes"
  - Single quote and double quote strings (with different properties)
  - Use of "default variables" in certain contexts.
  - Twisted syntax (e.g. "if(condition)" used as a suffix rather than prefix to conditionally executed code)
  - Parentheses around function arguments (may be) optional
  - ...

## Perl core

- Scalar variables
  - Usual naming rules, letter followed by letters, digits, underscores
  - '\$' type identifier
    - \$Temp
    - \$Counter1
  - **No explicit declaration**
  - **No fixed type** (reuse for integer, string etc)

*No explicit declaration or type: this is common in interpreted languages*

## Perl numerics:

- Integer
  - 12345
  - -6789
  - 0x1ac2
  - 0377 (octal)
- Real
  - 1.732
  - 1.21E2 (one hundred and twenty one)
  - 1.4E-2 (0.014)

## Numeric operators

- Usual: +, -, \*, /, % ; divide uses real arithmetic  
exponentiation operator \*\*  
++, ---
- Comparisons use
  - ==
  - !=
  - <
  - <=
  - >
  - >=

## Perl strings

- Two types:
  - Single quote strings
  - Double quote strings
- And some helper "functions" for declaring strings containing quote marks and for creating groups of strings.

## Perl strings 2

- Single quote string
  - Some oddities with backslash
  - Can continue over more than one line
    - 'Hi mum'
    - 'Hello World'
    - '\tHello World\n' (backslash, t, H, e, ..., backslash, n)
    - 'He said "hello world"'
    - 'Don\'t say "hello world"'
    - 'Backslashitis,\, hits again'

## Perl strings 3

- Double quoted strings
  - Backslash escapes similar to C (some additions)
    - \n, \r, \t, \f, \b (backspace), \a (bell) \e (escape)
    - \000 (octal values)
    - \xaf (hex value)
    - \cC, \cX etc (control characters)
    - \\, \'
    - \L ... \E, \U ... \E, \Q ... \E (case set)
  - "Interpolation" of variables

*Slight detour, string interpolation example ...*

*With side comments on uninitialized variables etc*

## Perl code fragment : *string interpolating scalar value*

```
#!/usr/bin/perl

$Days = 365;
$Hours = 24;
$Minutes = 60;
$Seconds = 60;
$Number = $Days * $Hours * $Minutes * $Seconds;

print ( "There are $Number seconds in a year\n");
```

*Doubly quoted Perl strings can "interpolate" the values of scalars*

## Perl: example program

- Treated as "main", sequence of statements to be executed.
  - Semicolons as statement separators.
  - Free form layout
- ```
$Number = $Days *
           $Hours * $Minutes
           * $Seconds;
```

## Perl: caution: uninitialized variables

```
$Days = 365;
$Hours = 24;
$Minutes = 60;
$Seconds = 60;
$Number = $Days * $Hours * $Minutes * $Seconds;
```

```
print ( "There are $Number seconds in a
year\n");
```

- Result: "There are 0 seconds in a year"
- Uninitialized variable, default to zero, **no error**

## Perl: caution: use warnings

- *Always develop code with "warning option"...*
- perl -w Calendar.pl  
Name "main::Seconds" used only once: possible typo at Calendar.pl line 7.  
Name "main::Sconds" used only once: possible typo at Calendar.pl line 6.  
Use of uninitialized value at Calendar.pl line 7.  
There are 0 seconds in a year

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

## Resume core Perl : Strings

## Double quote strings

- As in C/C++/Java, must protect double quotes embedded in strings:  
"I said \"STOP\""

## Quote operators

- Backslash quoting of " and ' symbols can be tiresome, so there are operators to help build quoted strings:  
q(...) (or q...\ or q..., etc )  
qq(...)

- Example

```
$num = 5;
$str1 = qq(I'm sorry but here is the ${num}th "Hello World" example\n);
$str2 = q(I'm sorry but here it the ${num}th "Hello World" example\n);
print $str1;
print $str2;
```

*Group quote operator, qw, covered later with lists.*

## *Oh, yes, there is another way ...*

- Strings can be provided as "here" documents

```
$str = <<TEST;
```

Does this really work?

Ooh. It does.

```
TEST
```

- <<DELIMNAME

## String Operators:

- Concatenation: "." operator  
"hello " . "world"
- Repetition:  
"Stop! " x 5
- Comparisons
  - eq
  - ne
  - lt
  - gt
  - le
  - ge

## String matching

- Major strength of Perl is regular expression string matching
- Considered later

## Accessing characters in strings

- Bit clumsy
- Have to use `substr(,,)` to isolate a character sequence of specified length starting at specified position.

## Control structures

## Program structures etc

- Typical simple programs:
  - Simple main line, starting after `#!/bin/perl`
- Using Functions:
  - Can declare own functions (*later, mostly obvious but some oddities about arguments*)
  - Structure is then function declarations followed by main line
- Objects:
  - Sort of, ...
  - Hybrid style, procedural main line (and functions) invoking methods of objects

## Basic procedural structure

- Sequence of
  - Assignments
  - Function calls (and method invocations on objects)
  - Selection constructs
  - Iteration constructs

## Assignment

- Special assignment operators
  - Numeric
    - +=, -=, \*= etc as in C/C++
  - String
    - .= concatenating assignment

## Expressions & operators

- Usual operand operator operand combinations
- Most of operators are familiar and have precedence as in C/C++ (*as always, use parentheses to clarify your intent*); include
  - Function call, [ ] (subscripting),
  - Arithmetic +, -, \*, /, %, \*\*, ++, --, <, <=, etc
  - String ., x
  - Selection ( ) ? ... : ...
  - Bit operations (shifts, masking)
  - Assignment operators
  - (see reference book for relative precedence)

## Expressions and operators

- Oddities
  - Note || (logical or), or (logical or) – with different precedence!
  - Pattern match operators =~ !~
  - Range operators .. and ...
  - , =>

## Diamond operator <>

- Used when working with “file handles” (input and output streams)
- <STDIN> is a reference to standard input stream
- <> Read from files identified as command line arguments
- Default is to read from a stream line-by-line  
\$DATA = <>
- (Normally output to stdout just uses print “function call”)

## Selection

- No switch statement!
- Ifs
  - if() { ... }
  - if() { ... } else { ... }
  - if() { ... } elsif { ... } elsif { ... } else { ... }
- Plus some oddities (more details later):
  - open(INPUT, “data.txt”) || die(“Can’t open data.txt”);
  - \$filecount++ if (“-” eq \$tag);

## Iteration

- while(condition) block
  - (there is a “continue” associated with while loops, it adds a loop-action like that of “for”)
  - (and there is a “redo”)
- until(condition) block
- for(initialize; termination-test; loop-action) block
  - C/C++’s continue is written as next;
  - C/C++’s break is written as last;
- for *each element in a list* block

## while (...) { ... } continue { ... }

- It gives a “while” something equivalent to the “loop-action” of a “for”

```
while(condition1) {  
  ...  
  if(condition2) { next; }  
  ...  
  if(condition3) { last; }  
  ...  
}  
continue { ... done for next and ordinary loop ... }
```

## Function call

- A word on its own is normally interpreted as a call to a function with that name

- So when you forget you \$ scalar type tag and write `sum = sum + value;` Perl gets all upset with these misplaced function calls to `function sum` and `value` (typically, will mutter something like “unquoted string “sum” may clash with future reserved word”)

## Function declarations

- Later!
  - Have to meet arrays before we can understand how arguments get passed to functions.

## Perl core functions

## Perl core functions

- Perl has built-in a large set of functions that includes most of the things that you will find on Unix with [man 2](#) or [man 3](#)

- Oddity – parentheses around arguments are optional, (and you can get confusing results!)

```
print 1+2+4;      # Prints 7.  
print(1+2) + 4;  # Prints 3.  
print (1+2)+4;   # Also prints 3!  
print +(1+2)+4;  # Prints 7.  
print ((1+2)+4); # Prints 7.
```

- Use `perldoc perlfunc` to get more information on functions

## Perl core functions

- Functions for SCALARs or strings
  - `chomp`, `chop`, `chr`, `crypt`, `hex`, `index`, `lc`, `lcfirst`, `length`, `oct`, `ord`, `pack`, `q/STRING/`, `qq/STRING/`, `reverse`, `rindex`, `sprintf`, `substr`, `tr///`, `uc`, `ucfirst`, `y///`
- Regular expressions and pattern matching
  - `m///`, `pos`, `quotemeta`, `s///`, `split`, `study`, `qr//`
- Numeric functions
  - `abs`, `atan2`, `cos`, `exp`, `hex`, `int`, `log`, `oct`, `rand`, `sin`, `sqrt`, `srand`

## Perl core functions

- Functions for real @ARRAYs
  - pop, push, shift, splice, unshift
- Functions for list data
  - grep, join, map, qw/STRING/, reverse, sort, unpack
- Functions for real %HASHes
  - delete, each, exists, keys, values

## Perl core functions

- Input and output functions
  - binmode, close, closedir, dbmclose, dbmopen, die, eof, fileno, flock, format, getc, print, printf, read, readdir, rewinddir, seek, seekdir, select, syscall, sysread, sysseek, syswrite, tell, telldir, truncate, warn, write
- Functions for fixed length data or records
  - pack, read, syscall, sysread, syswrite, unpack, vec

## Perl core functions

- Functions for filehandles, files, or directories
  - -X, chdir, chmod, chown, chroot, fcntl, glob, ioctl, link, lstat, mkdir, open, opendir, readlink, rename, rmdir, stat, symlink, umask, unlink, utime
- Keywords related to the control flow of your perl program
  - caller, continue, die, do, dump, eval, exit, goto, last, next, redo, return, sub, wantarray
- Keywords related to scoping
  - caller, import, local, my, our, package, use

## Perl core functions

- Miscellaneous functions
  - defined, dump, eval, formline, local, my, our, reset, scalar, undef, wantarray
- Functions for processes and process groups
  - alarm, exec, fork, getpgrp, getppid, getpriority, kill, pipe, qx/STRING/, setpgrp, setpriority, sleep, system, times, wait, waitpid
- Keywords related to perl modules
  - do, import, no, package, require, use

## Perl core functions

- Keywords related to classes and object-orientedness
  - bless, dbmclose, dbmopen, package, ref, tie, tied, untie, use
- Low-level socket functions
  - accept, bind, connect, getpeername, getsockname, getsockopt, listen, recv, send, setsockopt, shutdown, socket, socketpair
- System V interprocess communication functions
  - msgctl, msgget, msgrcv, msgsnd, semctl, semget, semop, shmctl, shmget, shmread, shmwrite

## Perl core functions

- Fetching user and group info
  - endgrent, endhostent, endnetent, endpwent, getgrent, getgrgid, getgrnam, getlogin, getpwent, getpwnam, getpwuid, setgrent, setpwent
- Fetching network info
  - endprotoent, endservent, gethostbyaddr, gethostbyname, gethostent, getnetbyaddr, getnetbyname, getnetent, getprotobyname, getprotobynumber, getprotoent, getservbyname, getservbyport, getservent, sethostent, setnetent, setprotoent, setservent
- Time-related functions
  - gmtime, localtime, time, times



## Many "Hello Worlds"

```
#!/share/bin/perl
print "How many hellos do you want?";
$num = <STDIN>;
$str = "Hello World! " x $num;
print $str . "\n";
```

## Many Hello Worlds ...

```
$num = <STDIN>;
$str = "Hello World! " x $num;
```

- Read line of input, num is the string "5\n";
- Gets used in context where numeric value expected, convert string to numeric value.

```
print $str . "\n";
```

- String concatenation

## Greeting

```
#!/share/bin/perl
print "What is your name?";
$name = <STDIN>;
chomp($name);
print "Hello $name, welcome to Perl\n";
```

- Input of name Fred would produce string "Fred\n", this would disrupt output line.
- `chomp()` removes any trailing \n from a string that has been read.

## Redo "ls -l"

- (CSCI204 exercise)
- Program to read input derived from Unix command `ls -l`

```
-r-x--x--x  1 root  bin   20796 Jan  6  2000 acctcom
-r-x--x--x  37 root  bin   5256 Jan  6  2000 adb
lrwxrwxrwx  1 root  root    29 Nov 30  2000 cachefspack ->
..lib/fs/cachefs/cachefspack
drwxr-xr-x  2 root  bin    512 Jun 10 15:08 sparcv7
```

- Output is to be a list giving permissions in octal, identifying file/directory type, and printing name. (Ignores special items like "links", "sticky bits", "set user id" flags etc); also count of number of files and directories

```
511 file acctom
511 file adb
755 directory sparcv7
```

## First version of "lister.pl"

```
#!/share/bin/perl -w

$files = 0;
$directories = 0;

while($str = <STDIN>) {
    chomp($str);
    ...
    print $code1 . $code2 . $code3 . "\t${type}\t${name}\n";
}

print "$files files, and $directories directories\n";
```

## "lister.pl" structure

- Initialize
- While loop, reads next line of input from `stdin`, (either pipe from `ls -l` or redirect from a file produced via `ls -l`)  
at end of file will get an empty line "" which is interpreted as false – terminates while loop
  - Process line
- Print totals for files and directories

## "lister.pl"

```
while($str = <STDIN>) {
  chomp($str);
  $char = substr($str,0,1);
  if ($char eq "-") { $type = "file"; $files++; }
  elsif($char eq 'd') { $type = "directory"; $directories++; }
  else { next; }
  $code1 = 0;
  if("r" eq substr($str,1,1)) { $code1 +=4; }
  if("w" eq substr($str,2,1)) { $code1 +=2; }
  if("x" eq substr($str,3,1)) { $code1 +=1; }
  ...
  print $code1 . $code2 . $code3 . "\t${type}\t${name}\n";
}
```

## Processing input line

- ```
$char = substr($str,0,1);
if ($char eq "-") { $type = "file"; $files++; }
elsif($char eq 'd') { $type = "directory"; $directories++; }
else { next; }
```
- Use `substr(,,)` to access characters
  - Note "next" rather than C/C++'s `continue`
  - Pick out individual flags associated with owner, group, and others – checking for explicit permissions

## First version of "lister.pl"

```
while($str = <STDIN>) {
  ...
  $pos = rindex($str, " ");
  $name = substr($str, $pos+1);

  print $code1 . $code2 . $code3 . "\t${type}\t${name}\n";
}
```

- Find last occurrence of the substring " " (single space) in string, select substring starting at that point

## substr

- Multiple forms
  - `substr(str,offset)`
  - `substr(str,offset,length)`
  - `substr(str,offset,length,replacement)`
- And offset can be negative, which is interpreted as being relative to end of string.

*Uhm! Typical Unix hack function, with interface now rated as being example of bad design*

## Alternate version of "lister.pl"

- Different scheme to compute octal code
- Different print function

```
$code = 0;
for($i=1;$i<10;$i++) {
  $code *=2;
  if("-" ne substr($str,$i,1)) { $code++; }
}

$pos = rindex($str, " ");
$name = substr($str, $pos+1);

printf "%o%s" , $code , "\t${type}\t${name}\n";
```

## Compilation gotchas ...

```
for(int $i=1;$i<10;$i++)
```

- No
  - no type declarations!
- if("-" != substr(str,i,1)) code++;
- No
  - Wrong inequality operator
  - Missing \$ (*scalar type tag*) on variable names (Perl starts looking for functions called str, i, code)
  - "if" takes a block, not a statement

## printf

- Since Perl interpreter built on top of C and using C's stdio I/O library, printf is natural.

```
printf "%o%s" , $code , "\t${type}\t${name}\n";
```

- First argument is a format string (here specifying an octal number and a string)
- Remaining data in comma separated list

## Of course, there is another way ...

- That printf format string is just a double quoted string and you can interpolate variables ...

```
printf "%o\t${type}\t${name}\n " , $code ;
```

## CSCI111: Fish and chips

- A new fish & chips restaurant called "UoW Fish & Chips" has just been opened, and the manager is looking forward to find a program that can satisfy his needs. Currently, the restaurant is only serving one type of dish, which costs \$ 5.95.

The customers can only order the number of dishes they want, but they cannot specify any other type of dish. This service will be improved at a later stage by the manager.

## CSCI111: Fish and chips : 2

- The manager wants to use a computer program to perform the following tasks:
  - A customer must be able to order directly, specifying how many dishes he/she wants to buy. The program must respond with the cost of the order and the order number (an integer value). Every time the program is run, the order numbers start from 1 again.
  - The manager wants to know what the maximum number of dishes was that was ordered by a customer in a single order. This figure must be reported in the total transaction report. The total transaction report must also indicate the total cost of all the orders.  
The total sales report can always be produced at any time whenever the manager wants to check the report.
  - The program must be driven with a simple menu that will make its operation easier.

## CSCI111: Fish and chips : 3

Welcome to UoW Fish & Chips Restaurant

1. Make an order
2. Print the total order
3. Quit

Enter your choice: <1>

How many dishes do you want to order? <1>

You are customer number: [1]

Your balance: \$[ 5.95]

Welcome to UoW Fish & Chips Restaurant

1. Make an order

...

Enter your choice: <2>

There was [1] customer

Total amount: \$[ 5.95]

Maximum number of dishes ordered in a single order: [1] dish

## FishShop.pl

```
#!/share/bin/perl -w

$cost = 5.95;
$orderNum = 0;
$maxorder = 0;
$totalSales = 0;
print "CSCI111: Fish and Chip Order Program";
while(1) {
    ...
}

print "Goodbye!\n";
```

## FishShop.pl

```
while(1) {
    print "Welcome to UoW Fish and Chips Restuarant";
    1. Make an Order
    2. Print totals
    3. Quit
    Enter your choice:;

    $order = <STDIN>;
    if ($order == 1) { ... }
    elsif($order == 2) { ... }
    elsif($order == 3) { last; }
    else { print "That was a bad choice. ... Try again"; }
}
```

*Take advantage of single quote string!*

*No switch statement!*

## FishShop.pl

```
while(1) {
    if ($order == 1) {
        print "How many dishes to you want to order?";
        $size = <STDIN>;
        chomp($size);
        if($size <= 0) { next; }
        $orderNum++;
        print "You are customer number : $orderNum\n";
        $ordercost = $cost * $size;
        print "Your balance: \$$ordercost\n";
        $maxorder = ($size > $maxorder) ? $size : $maxorder;
        $totalSales += $ordercost;
    }
    elsif ...
```

## FishShop.pl

```
    }
    elsif($order == 2) {
        print "There were $orderNum customers\n";
        print "Total amount: \$$totalSales\n";
        print "Maximum number of dishes ordered in a single
order: $maxorder dishes\n";
    }
    elsif($order == 3) { last; }
    else { print "That was a bad choice. ... Try again"; }
}

print "Goodbye!\n";
```

## Beyond CSCI111!

## Perl: Lists and arrays

## List (dynamic array) structure

- Perl has its own built in "collection class".
- Essentially a dynamic array:  
"does ..."
  - Create, usually initializing it with a non-empty set of elements
  - Add elements "at front" or "at end"
  - Remove elements "at front" or "at end"
  - Access elements at specific positions in the array
  - Get length of array
  - Copy array into another array
- Functions to
  - create a copy of an array with reversed set of elements
  - Sort an array (assuming contents are string values)

## @ type qualifier tag & *list literals*

- @varname  
varname is the name of an array.
- *List literals ...*
  - (1, 2, 3)
  - ("tom", "dick", "harry", "sue")
  - (\$name, \$address, \$town)
- List literals often used to initialize arrays

## Array creation examples

```
@PlacesIveBeen = ();  
@GradePts = (45, 50, 65, 75, 85);  
@TeenYears = ( 13 .. 19);  
■ Illustrates range  
■ Is the collection (13, 14, 15, 16, 17, 18, 19);
```

## More array creation

```
@Cities = ( "London", "Paris", "New York", "Rome",  
"Tokyo", "Sydney");  
■ Or maybe  
@Cities = qw (London Paris 'New York' Rome Tokyo  
Sydney);  
■ Er no  
■ (London, Paris, 'New, York', Rome, Tokyo, Sydney)  
■ But if you do have simple words, then qw is better
```

## There is always more than one way ...

```
#!/share/bin/perl -w  
  
@Cities1 = ("London", "Paris", "New York");  
@Cities2 = qw( Rome 'Los Angeles' "San Francisco");  
@Cities3 = ("Wagga", "Hay", "Cooma");  
@Cities4 = qw\ Thiroul Bellambi Keiraville\  
  
print "Cities1 :\n";  
$size = @Cities1;  
for($i=0;$i<$size;$i++) {  
    print $Cities1[$i] , "\n";  
}
```

## There is always more than one way ...

```
...  
print "Cities2 :\n";  
foreach $city (@Cities2) {  
    print $city , "\n";  
}  
print "Cities3 :\n";  
foreach $i (0 .. $#Cities3) {  
    print $Cities3[$i] , "\n";  
}  
print "Cities4 :@Cities4\n";
```

## Arrays: size, indexing

- Size and indexing

```
$size = @Cities1;
for($i=0;$i<$size;$i++) {
    print $Cities1[$i] , "\n";
}
```

- Scalar = array:

- *It means return size of array*

- Subscripting

- Scalar type tag, array name, subscript [ ] operator  
*(negative subscripts are permitted, work back from end of array)*

## foreach element in array ...

- foreach loop:

```
foreach $city (@Cities2) {
    print $city , "\n";
}
```

- Variation using range operator to generate sequence

```
foreach $i (0 .. $#Cities3) {
    print $Cities3[$i] , "\n";
}
```

- **`$#arrayname`**

- Means return index of last element in array

## Interpolation into a double quote string ...

- Array can be interpolated into a double quote string ...

```
print "Cities4 :@Cities4\n";
```

- `print @Cities4;`

- Prints contents of array – as continuous sequence

- `print "@Cities4";`

- Prints contents of array – space separated

## Splitting text into an array ...

- Many simple databases and spreadsheets have an option that lets you get a listing of their contents as a text file

- One line for each record

- Fields in record separated by a specific character (tab, or :)

- Example (name, role, dept, room, contact phone)

```
J.Smith:Painter:Buildings & Grounds::3456
T.Smythe:Audit clerk:Administration:15.205:3383
A.Solly:Help line:Sales:8.177:4222
```

## Splitting text into an array ...

- Need to be able to read such records and break out the various data fields.

- Use `split` function

- *Makes use of regular expression matching to find where to split string, but usually very simple in form because typically the split is done at a single marker character (the tab or colon separator character)*

## NoRoomPersons.pl

```
#!/share/bin/perl -w
```

```
while(<STDIN>) {
    @line= split /:/ ;
    # Line contains name, role, department, room, contact phone
    # Want to know which employees don't have a fixed work room
    # (room field should be empty)
    $room = $line[3];
    if(!$room) {
        print $line[0], "\n";
    }
}
```

## Yeah, some Perl tricks used

```
while(<STDIN>) {  
  @line= split /:/ ;  
  ■ Read in a line, but didn't assign it to a  
  variable!  
  It got assigned to the "anonymous" variable  
  $_ by default.  
  ■ @line array is being filled with output of split  
  breaking something at colon characters, what  
  something?  
  $_ is implicit
```

## Breaking lists apart

- In the last example, built arrays (lists) that contained name, role, department, etc.
- Can access using subscripting.
- But, if using data extensively might be useful to extract individual elements into scalar variables:

```
$name = $line[0];  
$role = $line[1];  
$department = $line[2];  
...
```

## There is another way ...

```
($name, $role, $department) = @line;  
■ Can use a list as a "lvalue" with an  
assignment from an array.  
■ Elements of the array are used to fill the  
scalars.  
■ List can contain an array (as its last member),  
this would get any otherwise unclaimed data  
values:  
($name, $role, $department, @rest) = @line;
```

## Could even do it this way ...

```
while(<STDIN>) {  
  ($name, $role, $department, $room, $phone)  
  = split /:/ ;  
  if(!$room) {  
    print $name, "\n" ;  
  }  
}
```

*Note: would get warnings about variables used only once*

## No lists of lists ... (this isn't LISP)

```
@Male = qw(Mickey Donald);  
@Female = qw(Minnie Daisy);  
@DisneyMob = (@Male, @Female, "Pluto");  
■ Produces a single level list:  
Mickey, Donald, Minnie, Daisy, Pluto
```

## "Slices" of arrays

- @line = qw(one two three four);
- @firsttwo = @line[0,1];
- @line[0,1] = ("five", "six");

## Pushing, popping, ...

```
■ push(array,value), pop(array)
@stack = ();
push(@stack, "one");
push(@stack, "two");
push(@stack, 3, 4);
push(@stack, 5, 6, 7, 8, "nine", 101 );
print @stack, "\n";
$val = pop(@stack);
print "@stack\n";
```

## ... and shifting

- unshift is similar to push, but puts things at start rather than end.
- shift is like pop, but removes item at start.

## Reverse, sort, and chomp

```
■ @newRevList = reverse(@aList);
■ @sortList = sort(@aList);
  ■ sort(qw(one two three four ))
    3 four one two
■ chomp(@list)
  ■ Removes "\n" terminating character from each element where present
  ■ Possibly useful if read an entire file into a list:
    @text = <STDIN>;
```

## sorting

- **Basic use of sort function:**  
@list1 = qw(This is a test what else Hello World Hi mom etc etc);  
**List1: This is a test what else Hello World Hi mom etc etc**  
**Sorted: Hello Hi This World a else etc etc is mom test what**  
@list2 = ( 100, 26, 3, 49, -11, 3001, 78);  
**List2 100 26 3 49 -11 3001 78**  
**Sorted: (default sort) -11 100 26 3 3001 49 78**  
**?!?!**
- But can modify sort behavior by providing own sort helper subroutine

## Sort helper subroutines

```
■ Kind of "built in", its style not representative of normal subroutines
■ Use "package global variables" $a and $b (two elements from the data set whose values are two be compared)
■ Example: numeric sort
sub by_number {
  if($a < $b) { return -1; }
  elsif($a == $b) { return 0; }
  else { return 1; }
}
```

## Sort program with subroutine

```
#!/share/bin/perl -w
sub by_number { ... }
@list1 = qw(This is a test what else Hello World Hi mom etc etc);
@list2 = ( 100, 26, 3, 49, -11, 3001, 78);
@slist1 = sort @list1;
print "List1 @list1\n";
print "Sorted List1 @slist1\n";
@slist2 = sort @list2;
print "List2 @list2\n";
print "Sorted List2 (default sort) @slist2\n";

@nlist2 = sort by_number @list2;
print "Sorted List2 (numeric sort) @nlist2\n";
```

## Output from program ...

```
List1 This is a test what else Hello World Hi mom etc etc
Sorted List1 Hello Hi This World a else etc etc is mom test what
List2 100 26 3 49 -11 3001 78
Sorted List2 (default sort) -11 100 26 3 3001 49 78
Sorted List2 (numeric sort) -11 3 26 49 78 100 3001
```

## There is always another way ... to sort

- Perl has a magical  $\Leftrightarrow$  operator for numeric comparisons
- Could write sub by\_number as

```
sub by_number {
    $a <=> $b
}
```
- *(Implicit return of value of final statement)*
- Could even have just:

```
@nlist2 = sort { $a <=> $b } @list2;
```

## List context, scalar context

- List used in "scalar" context is interpreted as request for list length:

```
$length = @mylist;
...
if(@inputlist) {
    #inputlist has elements ...
    ...
}
...
print @alist;
print scalar(@alist);
```

## "Keyword in context"

Generating permuted indexes of  
film titles

## Example of lists (arrays) and a subroutine

- Program is to:
  - **Read film titles**
    - Each title is a sequence of words, "keywords" start with a capital letter:  
The Empire Strikes Back  
The Return of the Jedi  
Moulin Rouge  
Picnic at Hanging Rock
  - **Produce a permuted keyword index**

```
The Empire Strikes Back
The Empire Strikes Back
Picnic at Hanging Rock
The Return of the Jedi
Moulin Rouge
Picnic at Hanging Rock
The Return of the Jedi
```

## Permuted title list generator

- *Note: this could be done more efficiently by simply searching in string for words; this example more to illustrate Perl features like "split".*
- **Approach**
  - Loop reading and processing each line of input
    - Find each "keyword" (capital letter start)
    - Generate a string with permuted context for keyword
    - Add string to collection
  - Sort the collection (using specialized sort helper subroutine)

## Line in permuted index

- Each line (string) will consist of
  - Start part (words preceding keyword)
    - Printed right justified
  - Keyword to end part
    - Printed left justified
- Line can be composed using sprintf:

```
$line = sprintf "%50s %-50s\n", $start, $end;
```

## Generating lines for keywords

- *This is the "inefficient" part.*
- Use split to break input line at word boundaries, get an array of words
- For each word in array
  - if word starts with capital letter generate a line
- Start part?
  - Accumulate as work along line
- End part?
  - Rest of word array from current point

## Example

- The Return of the Jedi
  - The
    - Start is ""
    - The is keyword
    - End is The Return of the Jedi
  - The Return
    - Start is "The "
    - Return is keyword
    - End is Return of the Jedi
  - The Return of
    - Start is "The Return "
    - of is not keyword

## Sorting the strings

- Building a collection of strings of length approx 104 characters (*50 right justified start, spaces, 50 left justified end*)
- Don't want normal sort function.
- Want strings sorted by the substring that begins at position 50.

## Sort helper subroutine

- Uses local variables!
  - New Perl construct "my" : used to declare variables local to subroutine (or inner block)
- ```
sub by_keyst {  
  my $str1 = substr($a,50);  
  my $str2 = substr($b,50);  
  if($str1 lt $str2) { return -1; }  
  elsif($str1 eq $str2) { return 0; }  
  else { return 1; }  
}
```

## FilmKwic.pl

```
#!/share/bin/perl -w  
  
sub by_keyst { ... }  
  
@collection = ();  
while($title = <STDIN>) {  
  ...  
}  
@sortcollection = sort by_keyst @collection;  
foreach $entry (@sortcollection) {  
  print $entry;  
}
```

## FilmKwic.pl

```
while($title = <STDIN>) {
  chomp($title);
  @Title = split // , $title;
  $start = "";
  foreach $i (0 .. $#Title) {
    $Word = $Title[$i];
    if($Word =~ /^[A-Z]/) {
      ...
    }
    $start .= $Word . " ";
  }
}
```

## Our first real string "regular expression" match!

```
if($Word =~ /^[A-Z]/)
```

- =~ regular expression match operator
  - Tests if lvalue (\$Word) matches pattern
- The pattern */stuff*
  - ^ this ^ means anchor test pattern at start of string
  - [A-Z] match any single upper case letter
- Means *Does Word start with a capital letter?*

## FilmKwic.pl

```
if($Word =~ /^[A-Z]/) {
  $end = "";
  for($j=$i;$j<=#Title;$j++)
    { $end .= $Title[$j] . " "; }
  $line =
    sprintf "%50s %-50s\n", $start, $end;
  push(@collection, $line);
}
```

## There is always another way ... building the list

```
$end = "";
for($j=$i;$j<=#Title;$j++)
  { $end .= $Title[$j] . " "; }
■ Or
$end = join ' ' $Title[$i .. $#Title];
```

## subroutines

## Perl subroutines

- sub name block
  - Has a return value
    - Value of last statement
    - Explicit `return`
  - Can have arguments
    - Gets arguments as the list (array) named `@_`
  - Has local variables
    - Use `my` qualifier to define them with correct scope

## Subroutine calls

- ```
Foo($arg1, $arg2, $arg3);
```
- Or
- ```
Foo $arg1, $arg2, $arg3;
```
- Arguments are simply assembled into a list (array): (`$arg1 $arg2 $arg3`)
  - An array argument is merged in with others (remember: no lists of lists)

## Extracting the arguments

- In the subroutine, start by separating the elements of the `@_` into individual local variables.  
Use a list lvalue ...
- ```
my($val1, $val2, @val3) = @_;
```
- `val1` is first argument, `val2` is second, `val3` would be a list with all remaining arguments

## Example: octal code

- Remember earlier example where wanted access permissions changed from string like `drwxr-x---` to an octal code like `750`

```
sub octal {  
  my $str = $_[0];  
  my $code = 0;  
  for(my $i=1;$i<10;$i++) {  
    $code *=2;  
    $code++ if("-" ne substr($str,$i,1));  
  }  
  return $code;  
}
```

## Octal code example

```
#!/share/bin/perl -w  
  
sub octal { ... }  
  
@codes = qw(drwxr-x--- -rwxr-xr-x -r----- -rw-rw-r--);  
  
for $code (@codes) {  
  printf "%s\t%o\n", $code, octal($code);  
}
```

## Example: member(item,list);

- Simple subroutine: determine whether list contains an entry equal to specified item (*as in "string equality"*).

```
sub member {  
  my($entry,@list) = @_;  
  for $memb (@list) {  
    if($memb eq $entry) { return 1; }  
  }  
  return 0;  
}
```

## Program using member subroutine

```
#!/share/bin/perl -w  
  
sub member { ... }  
  
@classrole = qw(tom dick harry sue robert fred jennie peter mark  
  carol ann anne);  
  
if(member("James", @classrole)) { print "James is here\n"; }  
else { print "James is wagging school again\n"; }  
  
$who = "ann";  
  
print "$who is here\n" if(member($who,@classrole));
```

## Actually, there is another way ...

- Didn't need to invent "member()" function as Perl has a general purpose version.
- **grep**
- `grep match_criterion datalist`
  - When used in list context, produces a sublist with references to those members of datalist that satisfy test
  - When used in scalar context, returns number of members of datalist that satisfy requirements

## grep

- Match criterion
  - Often is a regular expression that is matched against strings in list
  - Can be a block of code

```
if(grep { $_ eq $who } @classrole) { printf "$who is still here\n"; }
@littleones = grep { (length $_) < 4 } @classrole;
printf "The short names are @littleones\n";
@probablygirls = grep /[aeiou]$/ @classrole;
```

## Scope etc

## my local strict

- `my` introduces variable with limited scope, as in example subroutines.
- `local`
  - Establish name value binding in stack
  - A called subroutine needing value for a variable can climb stack until it finds a binding
- `strict`
  - Cut down on errors from mistyped variables
  - Requires even global (filescope) variable be declared prior to use

## Hashes

## Perl hash

- **Hash**
  - Uhm?
  - Really it is an associative array
  - Key/Value pairs
    - Key used as array index
    - Value is datum stored at that index
- Like a Perl list (array), a hash can grow as you add elements
- Easy to switch between an even length list and a hash (list is key0 val0 key1 val1 key2 ... valn)

## hash

- Perl uses % as type qualifier tag
  - %PostCode refers to complete hash structure called PostCode
- Mostly see hashes being used "in scalar context" where referring to individual key/value pairs
  - \$PostCode{"Wollongong"} = 2500;
  - \$PostCode{"Unanderra"} = 2526;
- Note the use of the { } brackets for indexing a hash

## Example hash:

- Cast list for a play
  - Three assumptions
    - There are sufficient actors in the company so that each actor plays only one part
    - Actors have distinct names
    - Roles have distinct names
- First Hash is %cast
  - Indexed by role name, value is actor name
- Second Hash is %players
  - Indexed by actor name, value is role name

## Hash example code

- Initialize cast hash
- Create players hash
  - %players = reverse %cast;
  - *(reverse may result in smaller sized hash! If two entries in original table had same value, i.e. one actor performs two roles, then only one will appear in reversed hash).*
- Look up values in the two hashes

## Hash example

```
#!/share/bin/perl -w

$cast{"First witch"} = "Angie";
$cast{"Second witch"} = "Karen";
$cast{"Third witch"} = "Sonia";
$cast{"Duncan"} = "Peter";
$cast{"Macbeth"} = "Phillip";
$cast{"Lady Macbeth"} = "Joan";
...
$cast{"Gentlewoman 3"} = "Holly";

%players = reverse %cast;
...
```

## Hash example

```
while(1) {
    print 'Enter command:
    1) Look up who plays role
    2) Look up role played by actor
    3) List roles
    4) Quit
    Command: ';
    $cmd = <STDIN>;
    if($cmd == 4) { last; }
    elsif($cmd == 3) { ... }
    elsif($cmd == 2) { ... }
    elsif($cmd == 1) { ... }
    else { print "Command not recognized\n"; }
}
```

## Hash example ...

```
elsif($cmd == 3) {
    @roles = keys %cast;
    for $role (@roles) { print $role, "\n"; }
}

■ keys hash
    ■ Returns a list of the keys in the hash
```

## Hash example ...

```
elsif($cmd == 2) {
    print "Enter actor's name : ";
    $Player = <STDIN>;
    chomp($Player);
    $Role = $players{$Player};
    if($Role) { print "$Player plays the role $Role\n"; }
    else { print "$Player is resting for this production\n"; }
}
elsif($cmd == 1) {
    print "Enter role name : ";
    $Role = <STDIN>;
    chomp($Role);
    $Actor = $cast{$Role};
    if($Actor) { print "$Role is being performed by $Actor\n"; }
    else { print "$Role not recognized as part of cast\n"; }
}
```

## Key not found?

- If key used when accessing a hash does not in fact occur  
\$theHash{\$badKey}  
then get value `undef` returned.
- In boolean context, `undef` is false hence the checks in the code.
- Can test explicitly for whether a scalar is defined  
if( `defined $val` ) { ... }
- Can ask a hash if it contains a given key:  
if( `exists $theHash{queryKey}` ) { ... }

## *Of course, there are other ways to initialize that hash ...*

- Setting each element is tiresome.
- Can create from a list

```
@cast = ("First witch", "Angie", "Second witch", "Karen", "Third
witch", "Sonia", "Duncan", "Peter", "Macbeth", "Phillip",
...
"Banquo", "John", "Lady Macduff", "Lois", "Porter",
"Neil", "Lennox", "Wang", "Angus", "Ian", "Seyton",
"Jeffrey", "Fleance", "Will", "Donaldbain",
...
"Gentlewoman 3", "Holly");
%cast = @cast;
```

*If your list does not have an even number of entries, the last key is given the value undef*

## *Or you can do it like this ...*

- Lists that long get unreadable and you are likely to mess up the pairings of keys and values.
- So *there is another way*:  
%cast = ("First witch" => "Angie",  
...  
"Donaldbain" => "Willy",  
"Menteith" => "Tim",  
...  
"Gentlewoman 3" => "Holly");

## Hash keys, Hash values

- To get a list of all keys in a Hash use keys  
`@roles = keys %cast;`
- To get a list of all the values in a Hash use values  
`@company = values %cast;`

## each Hash

- Hash structures basically have an Iterator associated with them.
- Can invoke `each` on a Hash
  - In list context, `each` returns a key/value pair as a two element list
  - In scalar context, `each` returns a key
  - Get pairs starting at beginning of hash and working to end
  - Then get an empty list
  - Then, if ask again, start iterator over again at beginning.

## Example: values, each, ...

```
3) List roles
4) List performers
5) Cast List
6) Quit
Command: ;
$cmd = <STDIN>;
if($cmd == 6) { last; }
elseif($cmd == 5) {
    print "Role                Played by\n";
    while(($r, $p) = each(%cast)) {
        printf "%-40s %-40s\n", $r, $p ;
    }
}
```

## Example: values, each, ...

```
$cmd = <STDIN>;
if($cmd == 6) { last; }
elseif($cmd == 5) {
    ...
}
elseif($cmd == 4) {
    @playlist = values %cast;
    @playlist = sort @playlist;
    for $person (@playlist) { print $person, "\n"; }
}
```

## delete

- You can remove entries from a hash with delete:  
delete \$theHash{theKey};

## Slice of hash

- Replace multiple values:  
@cast{"First witch", "Second witch", "Third witch" } =  
("Gina", "Christine", "Leila");

## Example with Hash and List



## WordCounter.pl

- Write a program that will count the number of occurrences of all distinct words in a document and will print a sorted list of these counts.
  - Words are to be folded to lower case forms
  - A word is any sequence of alphabetic characters
  - All non-alphabetic characters are to be ignored.

## WordCounter algorithm

- Loop reading lines from input
  - Split each line into words (split at any non-alphabetic character)
  - Force word to lower case
  - Use word as index into hash, increment value associated with this index
- Get a sorted list of keys of hash (i.e. a sorted list of words)
- For each entry in sorted list, index into hash to get count, print word and count.

## WordCounter.pl

```
#!/share/bin/perl -w
while($line = <STDIN>) {
    @words = split /[^\A-Za-z]/, $line;
    foreach $word (@words) {
        $index = lc $word;
        $counts{$index}++;
    }
}
@sortedkeys = sort keys %counts;
foreach $key (@sortedkeys) {
    print $key, "\t: ", $counts{$key}, "\n";
}
```

## WordCounter

```
@words = split /[^\A-Za-z]/, $line;
```

- Pattern specifies not (^) any upper case letter (A-Z) or any lower case letter (a-z), i.e. any non-alphabetic character.

```
$index = lc $word;
```

- lc is standard function, force to lower case
- `$counts{$index}++`;
- If \$index doesn't occur (previously) in %counts, then its value is "undef". In numeric context, "undef" is 0. So, on first encounter with word create an entry in hash with value 1. Subsequently, increment the value.

## *But I wanted them sorted by frequency ...*

- Try

```
foreach $key (sort
{ $counts{$a} <=> $counts{$b} }
keys % counts) {
    print $key, $counts{$key}, "\n";
}
```

## Perl versus ...

- Compare that solution in Perl with other languages:
- Java?
  - StringTokenizer to get words
  - Hashtable for counts
  - Iterators
  - All the same, probably several times as much code
- C++
  - "STL maps" anyone?
- C
  - `int main() { ... /* roll your own */ ... }`

## Learn to use languages appropriately!

- Most string manipulation tasks that you encounter are likely to be handled better in Perl than in other languages that you have met.
- Chose the language appropriate to the task.



## Inputs ...

- <STDIN> read from stdin (usual Unix options for redirection from a file)
- <> read from file(s) specified on command line (if several files given, their contents are concatenated)
- Explicitly open a file

## Opening input files

- Files identified by FILEHANDLES (*by convention, the things used to access files are almost always given names consisting entirely of capital letters*)
- Files opened using open(handle, filename)  
open(INPUT, "data.txt");
- The open function returns a boolean success flag, hence ...

```
if(! open(INPUT, "data.txt")) {
    print "Couldn't open 'data.txt'";
    exit 1;
}
```

## DIE, Die, die, ...

- die is Perl's "print error message and exit" function;  
can be used anywhere it's needed but most commonly encountered (at least in text book examples) in association with file I/O

```
open(INPUT, "data.txt") ||
    die "Couldn't open 'data.txt'\n";
```

## die oddities

- die "message"  
prints  
message at *scriptname* line *xxx*
- die "message\n"  
prints  
message

## System error information

- Another Perl "special variable" (\$!) has its value set following any system error.
- Could get more useful information:  
open(INPUT, "data.txt") ||  
die "Couldn't open 'data.txt' because \$!\n";

## warn

- “warn” is related Perl function.
- Prints error message, but doesn't call exit.

## Using an input filehandle

- Simply use named filehandle where previously have had STDIN

```
#!/share/bin/perl -w
open(INPUT, "data.txt") ||
    die "Coldn't read data.txt";
while($line = <INPUT>) {
    @words = split /[^A-Za-z]/, $line;
    foreach $word (@words) {
        $index = lc $word;
        $counts{$index}++;
    }
}
```

## Output filehandles

- Output filehandles can be created that allow writing to a file, or appending to an existing file:

```
open(OUTPUT, ">report.txt") || die ...;
open(ERRORS, ">>errlog.txt") || die ...;
```

- An output filehandle can be used in print statement:  
print OUTPUT "Role                    Actor\n";

*Bit on archaic side – it's formatting for line printers;  
these have died out*

## formats

- Alternative to use of printf with complex format strings
- Formats let programmer “visualize” the way the output will appear.
- But a little dated – these things mattered when reports sent to line printers, nowadays ...

*Bit on archaic side – it's formatting for line printers;  
these have died out*

## formats

- Essentially “text templates”
  - Contain some canned text (e.g. field labels)
  - Contain fields for printing data
    - Print fields represented pictorially
      - @<<<< is a 4 character field for left justified text
      - @| | | | | | | is an 8 character center justified field
      - @>>>>>> is a 6 character right justified field

*Bit on archaic side – it's formatting for line printers;  
these have died out*

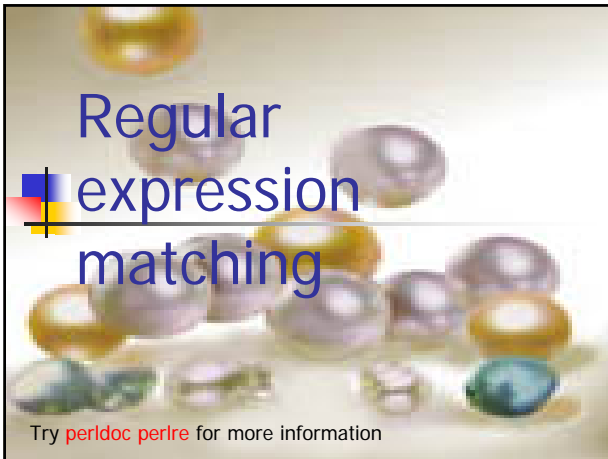
## fomat

- Declaration like

```
format OUTPUT =
Picture line
Argument line
Picture line
Argument line
...
```

- **Picture line:** canned text and field layouts
- **Argument line:** variables whose values are to be printed using preceding picture line.





## Regular expressions

- Regular expressions (regexes) define patterns of characters that can be matched with strings
  - Match a single character from this group of characters ...
  - Match one or more characters from this group, match a specified number (within the range ... to ...) of characters
  - Match any character not in this group
  - Restrict the match so that it must start at beginning of the string (or end at the end of the string)
  - Find a sequence that starts with ... then has ... then has ... and ends with ...
  - Split out the part of the string that matches ...

## Why regexes?

- Finding strings with particular patterns
  - String is sentence
  - Want to search for pair of words used in same sentence.
  - (This would be in an information retrieval system, use of words in context of single sentence often much more significant than their random occurrence somewhere in a document.)
- Extracting particular substrings from context
  - Context is HTML <a href="..." ... > link
  - Extract the value of the href
- Elaborate text rearrangements
  - Input is Pascal
  - Output is code with first stage of conversion to C

## How does Perl use them?

- Can match a string to a pattern with = ~
  - This returns true if the pattern was matched
- Can request substitutions
  - Replace anything of this form ... with this replacement ...
- Can get at the substrings that match specific subparts of a complex pattern

## Regex patterns

- Typically pattern represented as /stuff
- In the simplest patterns, stuff consists of the sequence of characters you wish to match:
  - /MasterCard/
  - /Bank Branch Number/
- However, many characters have specialized roles in defining more complex regular expressions; these characters must be "escaped" if you wish to match a literal string in which they appear:
 

```
{>[]0^$.|*+?
```

## Regex patterns

- You can use `m<delim> stuff <delim>` if you don't want the default `/stuff/` form for a pattern
 

```
m,Bank,
m!Credit!
```
- Can be useful e.g if you want to match file names
 

```
m#/usr/include/regex.h#
instead of
/\/usr\/include\/regex.h/
```
- Can have things like `\n`, `\t`, (and `\0172` etc) in patterns if trying to match these special characters.

## "Match" and "don't match" operators

- The operators are `=~` and `!~`
- Example

```
$line = <INPUT>;
if($line =~ /Bank Branch/) { ... }
```
- Special case, testing `$_`, just use the pattern:

```
while(<INPUT>) {
  if( /Quit/ ) { ... }
  ...
}
```

## Case insensitive matches ...

- Useful feature for checking inputs ...
- ```
while(<INPUT>) {
  if( /Quit/i ) { ... }
  ...
}
```
- Matches input "Quit" (canonical form) or quit (or qUIT or QUIT or qUit or ...)

## Alternatives

- Can specify alternative patterns within one regex

```
/MasterCard|Visa|AmEx/
```
- Can group alternatives in complex patterns

```
((cat's|dog's) (bowl|dish|plate))
```

*(should match a thing where you put your pet's food)*

Grouping has another (more important) role relating to retrieval of the parts of a string that match a pattern.

## Any character

- Character `."` matches any character (if you want to match a literal period character, you need `\.`)

## Character class

- Remember earlier example, wanted to match any vowel – i.e. a or e or i or o or u
- Can define a character class for vowels

```
[aeiou]
```
- Can use ranges in these definitions, e.g. the class of octal digits `[0-7]`, and hex digits `[0-9a-fA-F]`
- Can have a "negated" character class; start class definition with `^`

```
[^0-9]
```

 matches anything except a digit

## Perl's predefined character classes

- Character classes are part of most regular expression systems (which do also exist for C/C++ etc)
- Perl has some predefined character classes for commonly needed groups, these defined as special `\char` tags

|                   |                                   |                           |
|-------------------|-----------------------------------|---------------------------|
| ■ <code>\d</code> | digit                             | <code>[0-9]</code>        |
| ■ <code>\D</code> | negated <code>\d</code> ;         | <code>[^0-9]</code>       |
| ■ <code>\s</code> | whitespace                        | <code>[\ \t\r\n\f]</code> |
| ■ <code>\w</code> | (alphanumeric or <code>_</code> ) | <code>[0-9a-zA-Z_]</code> |

## Specifying number characters to match

- Character classes define match for a single character.
- Often want to specify that want some number of characters from that class
- Qualifiers on number of matches are placed after the character class definition
- Qualifiers are:
  - ? Optional tag, pattern to occur 0 or 1 times
  - \* Possible filler, pattern to occur 0 or more times
  - + Required filler, pattern to occur 1 or more times
  - {*n*} {*n*,} {*n*,*m*} Pattern to occur *n* times, or more, or the range *n* to *m* times

## Patterns with count qualifiers

- Spaces needed here* / /+
- 13 to 16 decimal digits* /0-9/{13,16}
- An optional + or – sign, one or more digits, an optional decimal point, optionally more digits*  
(+|-)?[0-9]+\.[0-9]\*

## Restricting matches to specified places in a string ...

- Can restrict a regex match so that
  - Must begin at start of line
  - Must end at end of line
  - Perl adds option to specify must begin (or end at a word boundary)
- Pattern that starts with ^ restricted to match at start of line
- Pattern that ends with \$ must match to end of line
- \b is Perl's special "word boundary" tag

## Perl variable substitution

- In some situations, Perl treats a regex pattern like a double quote string and will interpolate a value:

```
print "Enter keyword";
$keyword = <STDIN>;
while(<INPUT>) {
  if( /$keyword/ ) { ... }
```
- (Note how meaning of symbols like \$ and ^ depends a lot on where they appear in a pattern.)
- If you don't want variable substitution, and really want to match the string \$keyword, you can use m\$keyword
- If you want the substitution pattern precomputed, then use /\$keyword/o

## Example : simple regexes

- Helps those stuck doing a crossword puzzle.
- Enter a word pattern – letters where known, "." for unknown
- Program searches /usr/dict/words (Unix file with small dictionary of about 20000 words)

## Cheat.pl

```
#!/share/bin/perl

open(INPUT, "/usr/dict/words") || die;

print "Enter the word pattern you seek : ";
$wordpat = <STDIN>;
chomp($wordpat);

while(<INPUT>) {
  if( /^$wordpat$/ ) { print $_; }
}
```

## Cheat.pl

- Pattern `/^$wordpat$/`  
uses
  - `^` start at beginning of line
  - `$wordpat` Perl interpolation of value of string in scalar variable `wordpat`
  - `$` match to end of line (so matched word must be same length as user input)

## Cheat.pl

- User inputs:
  - `a...t` (*it's five letters, starting with a and ending with t*)
  - `a...[tf]` (*it's five letters, starting with a, I've got two ideas for "six-down" so this one might end in t or in f*)

## What matched?

- Often have input line containing filler text of no interest and various fields whose contents are required – *e.g. you want the numeric value of the string of digits that follow a dollar sign somewhere in the text.*
- To do this, use "groups" (defined by parentheses round a partial pattern)
- When match succeeds can ask for the text that matched each group in the pattern.

## Matched parts of string

- Look for \$ amount in string
- Pattern `^\$([0-9+)]\.\?([0-9]*)\D/`
  - `\$` a literal dollars sign
  - `([0-9+)]` a non-empty sequence of digits forming first group
  - `\.` An optional decimal point
  - `([0-9]*)` An optional sequence of digits forming second group
  - `\D` Any "non digit" character

## Dollar.pl

```
#!/share/bin/perl

while(1) {
    print "Enter string : ";
    $str = <STDIN>;
    if($str =~ /Quit/i) { last; }
    if($str =~ /\$([0-9+)]\.\?([0-9]*)\D/) {
        if($2) { $cents = $2; }
        else { $cents = 0; }
        print "Dollars $1 and cents $cents\n";
    }
    else { print "Didn't match dollar extractor\n"; }
}
```

## Testing dollar.pl

```
Enter string : This is a test of the $ program.
Didn't match dollar extractor
Enter string : This program cost $0.
Dollars 0 and cents 0
Enter string : This program should cost $34.99
Dollars 34 and cents 99
Enter string : QUIT
```

## Dollar.pl

- "Cents" field is optional, so \$2 may be empty
- ```
if($2) { $cents = $2; }  
else { $cents = 0; }
```
- (Can't have `if(! $2) { $2 = 0; }` because these \$ variables are effectively read only.)

## Matching "any" substrings.

- Quite often have a pattern like:
  - Some fixed text
  - A string whose value is arbitrary, but is needed for processing
  - Some fixed text
- Can use `.*` to match arbitrary string  
`/Fixed(.*)text/`

## Matching `.*`

```
while(1) {  
  print "Enter string : ";  
  $str = <STDIN>;  
  if($str =~ /Quit/i) { last; }  
  if($str =~ /Fixed(.*)text/) {  
    print "Matched with substring $1\n";  
  }  
  else { print "Didn't match\n"; }  
}
```

## Matching `.*`

Enter string : **Fixed up text** on slide.  
Matched with substring up

Enter string : **Fixed up this text**. Now  
starting to work on other **text**.

Matched with substring **up this text**. Now  
starting to work on other

## Greedy matching

- Matching arbitrary strings is sometimes problematic
- Pattern may match more than intended because matching is greedy – find the longest string that matches.
- Can change match to
  - `Fixed(.?)text`
  - `*?` (or `+?`) matches minimal sequence instead of normal maximal sequence

## Backreferences

- Sometimes want more complex patterns like:  
`fixed_text(somepattern)other_stuffSAMEPATTERNrest_of_line`
- Can achieve these matches using "back references" in the pattern string.
- Essentially same as referring to the matched substrings with `$1`, `$2` etc
- Use `\1` in pattern.

## Example with backreference

- Suppose you were preparing a Perl script that would automate the simple parts of a Pascal to C converter.
- You would want replacements like:
  - `Count := Count + 1;` => `Count++;`
  - `Count:= Count*Mul;` => `Count*=Mul;`
  - `Sum := Sum + 17;` => `Sum+=17;`

*Can do a large amount of such conversions via scripts.  
Not all. For example, Pascal has nested functions that have to be sorted out manually.*

## PascC

- Need a Pattern
  - Matches name (Lvalue) **\$1**
  - Matches Pascal's `:=` operator
  - Matches another name that is identical to first thing matched **\1**
  - Matches a Pascal `+, -, *, /` operator **\$2**
  - Matches either a number or another name **\$3**
  - Matches Pascal's terminating `;`
  - Allows extra whitespace anywhere

## PascC

- If didn't get match, just echo input
- If get match:
  - Print name of lvalue **\$1**
  - If form is `+ 1;` then output `++;`
  - If form is `- 1;` then output `--;`
  - Else print operator (**\$2**) = operand (**\$3**) .

## PascC.pl

```
while(1) {
    print "Enter string : ";
    $str = <STDIN>;
    if($str =~ /Quit/i) { last; }
    if($str A FAIRLY COMPLEX MATCH PATTERN!) {
        if(($3==1) && ($2 eq "+")) { print "\t$1++;\n"; }
        elsif(($3==1) && ($2 eq "-")) { print "\t$1--;\n"; }
        else { print "\t$1 $2= $3;\n"; }
    }
    else { print "$str\n"; }
}
```

## PascC.pl

- The pattern  
`/\s*([A-Za-z]\w*) *:= *\1 *(\+|\*|\-|\/) *(([0-9]+)|([A-Za-z]\w*)) *;/`
- First part  
`\s*([A-Za-z]\w*)`
  - Match leading whitespace characters
  - Define a group that
    - Starts with a letter and continues letters, digits, and underscores (Pascal variable name)
  - This is group 1 (**\$1** or **\1** depending on use)

## PascC.pl

- The pattern  
`/\s*([A-Za-z]\w*) *:= *\1 *(\+|\*|\-|\/) *(([0-9]+)|([A-Za-z]\w*)) *;/`
- Next part  
`*:= *\1`
  - Ignore whitespace
  - Match Pascal's `:=`
  - Ignore whitespace
  - Check for whatever we matched earlier (**\1**)

## PascC.pl

- The pattern  
`/\s*([A-Za-z]\w*)*:=*\1*(\+|\*|\w|-)*((\d+)|([A-Za-z]\w*))*/`
- Next part  
`*(\+|\*|\w|-)*`
  - Ignore whitespace
  - Match group 2 (\$2) which consists of a single Pascal operator +, -, \*, / (most of which need to be escaped)
  - Ignore whitespace

## PascC.pl

- Final part  
`((\d+)|([A-Za-z]\w*))`
  - Match group 3 which will be either
    - Non-empty sequence of digits
  - or
    - Pascal name (letter, followed by letters, digits, and underscore characters)

## Altering a string

- PascC.pl example essentially generated a new string based on string as input.
- Can actually use matching operators to change a string by substituting replacement text for text that is matched.

## Example with substitutions

- Again, for our Pascal to C converter, want to change Pascal's "not" operator to C's ! Operator (provided that we find it in suitable context).
- Looking for things like
  - if(**not** something) then ...
  - X := **not** something

## Substitution pattern

- `$str =~ s/(:=|\w)*not +\1 !/;`
- Change \$str if match the pattern.
- The pattern is
  - := operator or ( (group \$1, \1)
  - Optional spaces
  - **Not**
  - At least one space
- Replacement is
  - \1 (either := or )
  - ! C's not operator