Perl

A language for Systems and Network Administration and Management

Nick Urbanik

nicku@vtc.edu.hk

A computing department

What is Perl?

- Perl is a programming language

- The best language for processing text

- Cross platform, free, open

- Microsoft have invested heavily in ActiveState to improve sup
  for Windows in Perl

- Has excellent connection to the operating system

- Has enormous range of modules for thousands of application t

What is Perl? — 2

- Robust and reliable (has very few bugs)

- Supports object oriented programming

- Good for big projects as well as small

- Java 1.4 has borrowed one of Perl's best features: *regular ex
  sions*

- Perl has garbage collection

- The "duct tape of the Internet"

- Easy to use, since it usually "does the right thing"

- Based on freedom of choice: "There is more than one way t
  it!" — TIMTOWTDI™

Compiled and run each time

- Perl is interpreted, but runs about as fast as a Java program

- Software development is very fast

- The Apache web server provides `mod_perl`, allows Perl applications to run very fast

- Used on some very large Internet sites:

  - The Internet Move Database

  - Macromedia, Adobe, http://slashdot.org/

Perl is Evolving

- Perl 6 will introduce many great features to make Perl

  - easier to use

  - Even more widely usable for more purposes

  - Even better for bigger projects

Eclectic

- Borrows ideas from many languages, including:

- C, C++

- Shell

- Lisp

- BASIC

- . . . even Fortran

- Many others. . .

Regular Expressions

- One of the best features of Perl

- A new concept for most of you

- . . . But very useful!

- Used to:

  - extract information from text

  - transform information

  - You will spend much time in this topic learning about regular expressions — see slide 47

Why should I learn it?

- It will be in the final exam!

  - Okay, that's to get your attention, but. . .

- Consider a real-life sys-admin problem:

  - You must make student accounts for 1500 students

  - TEACHING BEGINS TOMORROW!!!

  - The Computing Division has a multi-million dollar application to give you student enrollment data

  - . . . but it can only give you PDF files with a strange irregular format for now (But Oh, it will be infinitely b in the future! Just wait a year or two. . .)

The available data

- Has a variable number of lines before the student data begins

- Has a variable number of columns between different files

- Has many rows per enrolled student

- Goes on for dozens of pages, only 7 students per page!!!!!!!

- There are two formats, both equally peculiar!!!!

Sample data for new courses:

```
15   N   CHAN Wai Yee              F 993175560  H123456(5)                 28210216              CHEUNG
         10-SEP-01 10-SEP-01                    21234567                                      WAI CHI
```

Problems

- There is a different number of lines above the student records

- There is a different number of characters within each column from file to file

- There are many files

- The format can change any time the computing division determines necessary

Solution in Perl — 1

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

use strict;

my $course;
my $year;

while ( <> )
{
    chomp;

    if ( /^\s*Course :\s(\d+)\s/ )
    {
        $course = $1;
        undef $year;
        next;
    }
```

Solution in Perl — 2

```perl
    elsif ( m!^\s*Course :\s(\d+)/(\d)\s! )
    {
        $course = $1;
        $year = $2;
        next;
    }
    if (
        my ( $name, $gender, $student_id, $hk_id )
            = m{
                \s\s+                   # at leaset 2 spaces
                (                       # this matches $name
                    [A-Z]+              # family name is upper case
                    (?:\s[A-Z][a-z]*)+  # one or more given names
                )
                \s\s+                   # at leaset 2 spaces
                ([MF])                  # gender
                \s+                     # at least one space
                (\d{9})                 # student id is 9 digits
                \s\s+                   # at leaset 2 spaces
                ([a-zA-Z]\d{6}\([\dA-Z]\)) # HK ID
            }x
    )
```

Solution in Perl — 3

```perl
    {
        print "sex=$gender, student ID = $student_id, ",
        "hkID = $hk_id, course = $course, name=$name, ",
        defined $year ? "year = $year\n" : "\n";
        next;
    }
    warn "POSSIBLE UNMATCHED STUDENT: $_\n" if m!^\s*\d+\s+!;
}
```

But I can use any other language!

- I will give you HK$200 if you are the first person to write a solution in another language in fewer keystrokes

- Note: the Perl solution given has:
  - comments
  - Plenty of space to show structure
  - ...and handles exceptional situations (i.e., it is robust)

- To claim your $200 from Nick, your solution must have
  - similar space for comments
  - Similar readability and robustness
  - Be written in a general purpose language using ordinary libraries

Other Solutions may take Longer to Write

- This program took a very short time to write

- It is very robust

- For problems like this, Perl is second to no other programming language.

The hello world program

```
print "hello world\n"
```

Variables

- There are three basic types of variable:

- **_Scalar_** (can be a number or string or...)

- **_Array_** (an ordered array of scalars)

- **_Hash_** (an unordered array of scalars indexed by strings ins of numbers)

- Each type distinguished with a "funny character"

$Scalars:

- Start with a dollar sign

- Hold a single value, not a collection

- A string is a scalar, so is a number

- Since Perl is a *loosely typed language*, a scalar can be an int a floating point number, a character or a string.
  - Note that later you will see that a scalar can also ho *reference* to another piece of data, which may also b array or hash.

- Examples:

  ```
  $apple = 2;
  $banana = "curly yellow fruit";
  ```

@Array

- Starts with a @

- Indexes start at 0, like in C or Java

- Each entry in an array is a scalar.

  - Multidimensional arrays are made by entry of an array being
    a reference to another array.

- See slide 19

%Hashes

- Unfamiliar concept to many of you

- Like an array, but indexed by a string

- A data structure like a database

- See slide 22

Conclusion

- Perl is optimised for text and systems administration program-
  ming

- Has great portability

- Is strongly supported by Microsoft

- Has three main built-in data types:

- Scalar: starts with $

- Array: starts with @

- Hash: starts with %

An Overview of Perl

A language for Systems and Network Administration and Management:

An overview of the language

Where do I get Perl?

- For Windows, go to http://www.activestate.com, download
  installer

- For Linux: it will be already installed

- For other platforms: go to http://www.perl.com

- This is a good source of other information about Perl

Where do I get Info about Perl?—1

- On your hard disk:

  - $ `perldoc -f` ⟨*function*⟩

    * will look up the documentation for the built-in ⟨*function*⟩ (from the documentation `perlfunc`)

  - $ `perldoc -q` ⟨*word*⟩

    * will look up ⟨*word*⟩ in the headings of the FAQ

  - $ `perldoc perl`

    * shows a list of much of your locally installed documentation, divided into topics

  - ActiveState Perl provides a Programs menu item that links to online html documentation

CPAN, PPM: Many Modules

- A very strong feature of Perl is the community that supports

- There are tens of thousands of third party modules for many, purposes:

  - Eg. `Net::LDAP` module supports all LDAP operat `Net::LWP` provides a comprehensive web client

- Installation is easy:

  $ `sudo perl -MCPAN -e shell`
  `cpan> install Net::LDAP`

- Will check if a newer version is available on the Internet CPAN, and if so, download it, compile it, test it, and if it pa tests, install it.

Where do I get Info about Perl?—2

- Web sites:

  - http://www.perl.com
  - http://www.activestate.com
  - http://use.perl.org

- See slide 64 for a list of books.

PPM: Perl Package Manager

- For Windows

- Avoids need for a C compiler, other development tools

- Download precompiled modules from ActiveState and other s and install them:

  `C:\> ppm install Net::LDAP`

- See documentation with ActiveState Perl

Mailing Lists: help from experts

- There are many mailing lists and newsgroups for Perl

- When subscribe to mailing list, receive all mail from list

- When send mail to list, all subscribers receive

- For Windows, many lists at http://www.activestate.com

How to ask Questions on a List

- I receive many email questions from students about many topics

- Most questions are not clear enough to be able to answer in any way except, "please tell me more about your problem"

- Such questions sent to mailing lists are often unanswered

- Need to be concise, accurate, and clear

- see also Eric Raymond's *How to Ask Questions the Smart Way* at http://catb.org/~esr/faqs/smart-questions.html

- Search the FAQs first—see slide 13

Where is Perl on my system?

- ActiveState Perl installs `perl.exe` in `C:\Perl\perl.exe`

- Linux systems have a standard location for perl at `/usr/bin/perl`

- On some Unix systems, it may be installed at `/usr/local/bin/perl`

How OS knows it's a Perl program—1

- To run your Perl program, OS needs to call perl

- How does OS know when to call Perl?

- Linux, Unix:

  - programs have *execute* permission:

    `$ chmod +x` $\langle program \rangle$

    * OS reads first 2 bytes of program: if they are "`#!`" read to end of line, then use that as the interpreter
    * OS doesn't care what your program file is called

  - If program file is not in a directory on your `PATH`, call it this:

    `$ ./`$\langle program \rangle$

How OS knows it's a Perl program—2

- Windows:

  - OS uses the extension of the file to decide what to do (`.bat`, `.exe`)

  - Your program names end with `.pl`

- For cross platform support:

  - Put this at the top of all your programs:

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

  - Name your programs with an extension `.pl`

Language Overview

- variables: scalars, arrays and hashes — §18–§27

- compiler warnings, `use strict;` — §26–§27

- operators, quoting — §28–§29

- input and output — §30

- statements: — §31

  - `if...elsif...else` and `unless` statements — §31–§32

  - `while`, `for` and `foreach` loops — §32–§36

    * iterating over arrays and hashes — §36–§37

  - Exit early from a loop with `last`, and `next` — §38

  - "backwards" statements — §38–§39

Funny Characters $, @, %

- Variables in Perl start with a *funny character*

- Why?

- No problem with reserved words:

- can have a variable called `$while`, and another variable c
  `@while`, and a third called `%while`.

- Can *interpolate* value into a *Double-quoted* string (but not a s
  quoted string):

  ```
  my $string = "long";
  my $number = 42.42;
  print "my string is $string ",
      "and my number is $number\n";
  ```

Language Overview — 2

- We also will examine:

  - subroutines, parameters and `return` statement — §41–§42

  - array operations — §39–§40

  - Error reporting: `die` and `warn` — §42

  - Opening files — §43–§44

  - executing external programs — §44–§46

  - regular expressions — §47–§60

  - Special input modes — §61–§62

  - One line Perl programs — §63

Arrays

- Define an array like this:

  ```
  my @array = ( 1, 5, "fifteen" );
  ```

- This is an array containing three elements

- The first can be accessed as `$array[0]`, second as `$array[1]`
  last as `$array[2]`

- Note that since each element is a scalar, it has the `$` funny c
  acter for a scalar variable *value*

- In Perl, *we seldom use an array with an index*—use list proces
  array operations: `push`, `pop`, `shift`, `unshift`, `split`, `grep`,
  and iterate over arrays with the `foreach` statement—see sli

  - higher level.

Array Examples

- Use the `qw//` "quote words" operator to help initialise arrays — see slide 29

- See slide 36 for how the `foreach` loop works.

```
my @fruit = qw( apple banana mandarin
                peach pear plum );
foreach my $fruit ( @fruit ) {
    print "$fruit\n";
}
```

- Note that these two are equivalent:

```
my @fruit = qw( apple banana mandarin
                peach pear plum );
my @fruit = ( "apple", "banana", "mandarin",
              "peach", "pear", "plum" );
```

More About Arrays

- Instead of initialiasing the array as in slide 19, we can initi the elements one by one:

```
my @fruit;
$fruit[ 0 ] = "apple";
$fruit[ 1 ] = "banana";
# ...
$fruit[ 5 ] = "plum";
```

- We can get a *slice* of an array:

```
my @favourite_fruit = @fruit[ 0, 3 ];
print "@favourite_fruit\n";
```

  – execute the program:

```
$ ./slice.pl
apple peach
```

List Assignment

- We can use a list of scalars whenever it makes some sense, e.

  – We can assign a list of scalars to a list of values

- Examples:

```
my ( @a, $b, $c ) = ( 1, 2, 3 );
my @array = ( @a, $b, $c );
my ( $d, $e, $f ) = @array;
```

Even More About Arrays

- How many elements are in the array? See slide 22

  ```
  print scalar @fruit, "\n"
  ```

- Does the array contain any data? See slide 32

  ```
  print "empty\n" unless @fruit;
  ```

- Is there any data at the index `$index`?

  ```
  if ( defined $fruit[ $index ]
       and $fruit[ $index ] eq "apple" ) {
     print "found an apple.\n";
  }
  ```

  - See `perldoc -f defined`. Also see `perdoc -f exists`.

Scalar, List Context

- Each part of a program expects a value to be either *scalar* or *list*

- Example: `print` is a list operator, so if you `print` something, it is in *list context*

- If you look in the *Perl Reference*, you will see LIST shown as a parameter to many functions.

  - Any value there will be in a *list context*

- Many built-in functions, and your own functions (see `perldoc -f wantarray`), can give a different result in a scalar or list context

- force scalar context with `scalar`, e.g.,

  ```
  print "the time is now ", scalar localtime, "\n";
  ```

Hashes

- Hashes are probably new to you

- Like an array, but indexed by a string

- Similar idea was implemented in `java.lang.HashTable`

- Perl hashes are easier to use

Initialising a Hash

```
my %hash = ( NL => 'Netherlands',
             BE => 'Belgium' );
```

- This creates a hash with two elements

- one is `$hash{NL}`, has value "`Netherlands`";

- the other is `$hash{BE}` with value "`Belgium`"

- The "`=>`" is a "*quoting comma*".

  - It is the same as a comma, but it also quotes the strin its left.

  - So you can write the above like this:

    ```
    my %hash = ( 'NL', 'Netherlands',
                 'BE', 'Belgium' );
    ```

    but the "`=>`" operator make it more clear which is the and which is the value.

Hash Examples — 1

- As with arrays, you make a new element just by assigning to it:

```perl
my %fruit;
$fruit{apple} = "crunchy";
$fruit{peach} = "soft";
```

- Here, we made two hash elements.

  - The keys were "apple" and "peach".

  - The corresponding values were "cruchy" and "soft".

- You could print the values like this:

```perl
print "$fruit{apple}, $fruit{peach}\n";
```

prints: crunchy, soft

Hash Examples — 2

- How to see if a hash is empty? See 32

```perl
print "empty\n" unless %fruit;
```

- How to delete a hash element?

```perl
delete $fruit{coconut};
```

- Hashes are often useful for storing counts (see slides 32–34 for more about while loops):

```perl
my %wordcounts;
while ( <> ) {
    chomp;
    ++$wordcount{$_};
}
```

Hash slices

- We can assign some values to part of a hash:

```perl
$score{fred} = 150;
$score{barney} = 100;
$score{dino} = 10;
```

- We could use a *list assignment* (see §21):

```perl
( $score{fred}, $score{barney}, $score{dino} )
    = ( 150, 100, 10 );
```

... too long. A *hash slice* makes this easier:

```perl
@score{ "fred", "barney", "dino" } = ( 150, 100, 10 );
```

- We can *interpolate* this too (see slides 18 and 29):

```perl
my @players = qw( fred barney dino );
print "scores are @score{@players}\n";
```

Another Hash Example

- Often used to keep a count of the number of occurrences of read in:

```perl
#! /usr/bin/perl -w
use strict;
our %words;
while ( <> ) {
    next unless /\S/; # Skip blank lines
    my @line = split;
    foreach my $word ( @line ) {
        ++$words{$word};
    }
}
print "Words unsorted, in the order they come from the hash:\n\n";
foreach my $word ( keys %words ) {
    printf "%4d %s\n", $words{$word}, $word;
}
```

- see slide 32 for while loop, slide 34 for while (<>), slide 36 for the foreach state

slides 32 and 38 for the unless statement

Hashes are Not Ordered

- A *big difference from arrays* is that hashes have *no order*.

- The data in a hash will be available in only an *unpredictable order*.

- See slide 36 for how to *iterate* over hash elements

use strict and Declaring Variables

- All programs that are more than a few lines long should have
  pragma use strict;

- This turns on additional checking that all variables are decl
  all subroutines are okay, and that references to variables are "
  references" — see perldoc strict.

- All variables that you use in your program need to be decl
  before they are used with either my or our.

- my defines a local variable that exists only in the scope of
  current block, or outside of a block, in the file.

  – See perldoc my.

- our defines a global variable.

  – See perldoc our.

Discipline—use warnings

- Better to let compiler detect problems, not your customer

- Develop your program with all warnings enabled

- Either:

  – put -w as an option to perl when execute the program, i.e.,

    * Make the first line of your program:
      #! /usr/bin/perl -w
    * Or better: put a line:
      use warnings;
      near the top of your program.

Examples of `use strict` and Variables

- Without `use strict`, a variable just springs into life whenever you use it.

- *Problem*: a typing mistake in a variable creates a *new variable* and a hard-to-find bug!

- ... so *always start your programs like this:*

  ```
  #! /usr/bin/perl
  use warnings;
  use strict;
  ```

- `use warnings;` enables compile time warnings which help find bugs earlier—see `perldoc warnings`

- After `use strict`, it will be an error to use a variable without declaring it with `my` or `our`.

  - Most code examples in these notes define variables with `my` or `our`

Operators and Quoting

- Perl has all the operators from C (and so Java), in same preced

- Has more operators for strings:

- Join strings with a dot, e.g.

  ```
  print "The sum of 3 and 4 is " . 3 + 4 . "\n";
  ```

- Quote special characters with backslash, as in C or Java

  ```
  print "\$value = $value\n";
  ```

- Can quote *all* characters using single quotes:

  ```
  print 'output of \$perl = "rapid";print \$perl; is "rapid"';
  ```

- Note that double quotes are okay in single quotes, single qu okay in double quotes.

- Documentation in `perldoc perlop`.

Quoting

- Perl has lots of ways of quoting, too many to list here

		Meaning	**Interpolates**	**Slide**
''	q//	Literal	No	§28, §18
""	qq//	Literal	Yes	§28, §18
``	qx//	Command	Yes	§46
()	qw//	quote word list	No	§19,§38
//	m//	Pattern match	Yes	§50
s///	s///	Substitution	Yes	§59
y///	tr///	Translation	No	

  - See slide 18 for meaning of "interpolate"

- y/// or tr/// works just like the POSIX tr (translate) program in Linux.

Input and Output

- Read from standard input like this:

```
my $value = <STDIN>;
```

- Note that there will be a newline character read at the end

  - To remove trailing newline, use chomp:

```
chomp $value;
```

  - The word STDIN is a predefined *filehandle*.

    * You can define your own filehandles with the open built-in function.

- write to standard output with the list operator print

  - print takes a list of strings:

```
print "The product of $a and $b is ",
      $a * $b, "\n";
```

What is Truth?

- Anything that has the string value "" or "0" is false

- Any other value is true.

- This means:

  - No number is false except 0

  - any undefined value is false

  - any reference is true (see perldoc perlref)

- Examples:

```
0      # becomes the string "0", so false
1      # becomes the string "1", so true
0.00   # becomes 0, would convert to the string "0", so false
""     # The null string, so false
"0.00" # the string "0.00", neither empty nor "0", so true
undef() # a function returning the undefined value, so false
```

Statements for Looping and Conditions

- We look at the following statements in the language:

  - if...elsif...else statements — §31

    * The unless statement is similar to the if statement §32

  - while loops — §32

    * processing input using while
    * The <> operator

  - for loops — §35

  - foreach loops — §36

    * iterating over arrays and hashes with foreach, while §36–§37

  - Exit early from a loop with last, and next — §38

- We will also look at "*backwards statements*" — §38–§39

`if` Statements

- `if` statements work as in C or Java, except:

  - braces are required, not optional

  - Use `elsif` instead of `else if`

- Example:

```
if ( $age > $max ) {
    print "Too old\n";
} elsif ( $age < $min ) {
    print "Too young\n";
} else {
    print "Just right\n";
}
```

`while` loop

- Just as in C or Java

  - . . . but braces are required:

```
while ( $tickets_sold < 1000 ) {
    $available = 1000 - $tickets_sold;
    print "$available tickets are available.  ",
            "How many do you want: ";
    $purchase = <STDIN>;
    chomp $purchase;
    $tickets_sold += $purchase;
}
```

-

`unless` Statement

- Same as `if` statement,

  - except that the block is executed if the condition is *false*:

```
unless ( $destination eq $home {
    print "I'm not going home.\n";
}
```

                    ↙   corresponds to: ↘

```
unless ( ⟨condition⟩ ) {      if ( ! ( ⟨condition⟩ ) ) {
    ⟨statements...⟩;              ⟨statements...⟩;
}                             }
```

- `else` works, but I suggest you don't use it

  - Use `if`. . . `else` instead

Input with `while`

- Input is often done using `while`:

```
while ( $line = <STDIN> ) {
    ⟨process this $line⟩
}
```

- This loop will iterate once for each line of input

- will terminate at end of file

The Special $_ variable

- *Nearly every built-in input function*, *many input operators*, *most statements with input* and *regular expressions* use a *special variable* $_

- If you don't specify a variable, *Perl uses* $_

- For example, this `while` loop reads one line from standard input at a time, and prints that line:

```
while ( <STDIN> ) {
    print;
}
```

- `while` loop reads one line into $_ at each iteration.

- `print` statement prints the value of $_ if you do not tell it to print anything else.

- See the *Perl Reference* on page 2 under *Conventions*

while and the <> operator

- Most input is done using the <> operator with a `while` loop

- The <> operator processes files named on the *command line*

  - These are called *command line parameters* or *command
    arguments*

  - If you execute it like this:

    `angle-brackets.pl`

    then you have no *command line arguments* passed to
    program.

  - But if you execute it like this:

    `angle-brackets.pl file_1 file_2 file_3`

    then the *command line* has three *arguments*, which here,
    pen to be the names of files.

while and the <> operator — 2

- We most often use the <> operator like this:

```
while ( <> ) {
    ⟨statements. . .⟩
}
```

- *This loop does a lot*. The pseudocode here shows what it doe

```
if there are no command line arguments,
    while there are lines to read from standard input
        read next line into $_
        execute ⟨statements...⟩
else
    for each command line argument
        open the file
        while there are lines to read
            read next line from the file into $_
            execute ⟨statements...⟩
        close the file
```

`for` loop

- The `for` loop works as in C or Java, except that braces are required, not optional.

- Example:

```
for ( $i = 0; $i < $max; ++$i ) {
    $sum += $array[ i ];
}
```

- Note that we rarely use this type of loop in Perl. Instead, use the higher level `foreach` loop...

`foreach` loop

- The `foreach` loop iterates over an array or list.

- Most useful looping construct in Perl

- It is so good, that Java 1.5 has borrowed this type of loop to simplify iterators.

- An example: adds 1 to each element of an array:

```
foreach my $a ( @array ) {
    ++$a;
}
```

- `$a` here is a *reference* to each element of the array, so

- changing `$a` actually changes the array element.

- You can write "for" or "foreach", Perl won't mind.

Iterating over a Hash

- Referring to our example hash in slide 22, we can process element like this:

```
foreach my $key ( keys %hash ) {
    ⟨process $hash{$key}⟩
}
```

   - `keys` creates a temporary array of all the keys of the ha

   - We then looped through that array with `foreach`.

- More efficient is to use the `each` built in function, which iterates through the hash:

```
while ( my ( $key, $value ) = each %hash ) {
    ⟨process $key and $value⟩
}
```

Iterating over a Hash in Sorted Order

- Did we process the contents of `%hash` in alphabetical orde slide 36?

   - No.

   - So what do we do if we want to print the elements in or

      * In order of key by alphabet? Numerically?

      * In order of element by alphabet? Numerically?

- Use built in `sort` function

- see `perldoc -f sort`

Iterating over a Hash in Sorted Order

- You *cannot sort a hash*

- ...but you can read all the keys, sort them, then process each element in that order:

```
foreach my $key ( sort keys %hash ) {
    ⟨process $hash{$key}⟩
}
```

  – see `perldoc sort`

- A reverse sort:

```
foreach my $key ( reverse sort keys %hash ) {
    ⟨process $hash{$key}⟩
}
```

  – see `perldoc reverse`

Exit a Loop Early

- Java and C provide `break` and `continue`

- Perl provides `last` and `next`

```
my @super_people = qw( Superman Robin
                       Wonder Woman
                       Batman Superboy );
foreach my $person ( @super_people ) {
    next if $person eq "Robin";
    print "$person\n";
    last if $person eq "Batman";
}
```

- What do you think this program will print?

"Backwards" Statements

- Put an `if`, `while` or `foreach` modifier *after a simple statem*

- You can put a simple statement (i.e., with no braces), and one of these afterwards:

```
if EXPR
unless EXPR
while EXPR
until EXPR
foreach EXPR
```

"Backwards" Statements—Examples

- Examples:

  – `print $1 if /(\d{9})/;`
    is equivalent to:

    ```
    if ( /(\d{9})/ )
    {
        print $1;
    }
    ```

  – `# print unless this is a blank line:`
    `print unless /^\s*$/;`
    is equivalent to

    ```
    if ( ! /^\s*$/ ) {
        print;
    }
    ```

Array Operations—push and pop

- The documentation for these is in the very loo–oong document perlfunc, and is best read with `perldoc -f ⟨Function⟩`

  push add a value at the end of an array, e.g.,

  ```
  my @array = ( 1, 2, 3 );
  push @array, 4;
  # now @array contains ( 1, 2, 3, 4 )
  ```

    – Do `perldoc -f push`

  pop remove and return value from end of an array

  ```
  my @array = ( 1, 2, 3 );
  my $element = pop @array;
  # now @array contains ( 1, 2 )
  # and $element contains 3
  ```

    – Do `perldoc -f pop`

Array Ops—shift and unshift

shift remove and return value from the beginning of an array, e.g.,

```
my @array = ( 1, 2, 3 );
my $element = shift @array;
# now @array contains ( 2, 3 )
# and $element contains 1
```

- Do `perldoc -f shift`

unshift add value to the beginning of an array, e.g.,

```
my @array = ( 1, 2, 3 );
unshift @array, 4;
# now @array contains ( 4, 1, 2, 3 )
```

- Do `perldoc -f unshift`

split and join

- Do `perldoc -f split` and `perldoc -f join`.

- split splits a string into an array:

  ```
  my $pwline
      = "nicku:x:500:500:Nick Urbanik:/home/nicku:/bin/bash";
  my ( $userid, $pw, $userid_number, $group_id_number,
      $name, $home_dir, $shell ) = split /:/, $pwline;
  ```

- Another application is reading two or more values on the s input line:

  ```
  my ( $a, $b ) = split ' ', <STDIN>;
  ```

- join is the opposite of split and joins an array into a strin

  ```
  my $pwline = join ':', @pwfields;
  ```

Subroutines

- See `perldoc perlsub`

- Syntax:

  ```
  sub ⟨subroutine name⟩
  {
      ⟨statements...⟩
  }
  ```

Parameters — 1

- Subroutines calls pass their parameters to the subroutine in an list named `@_`. It is best to show with an example:

```perl
#! /usr/bin/perl -w
use strict;
sub product
{
    my ( $a, $b ) = @_;
    return $a * $b;
}
print "enter two numbers on one line: a b ";
my ( $x, $y ) = split ' ', <STDIN>;
print "The product of $x and $y is ",
    product( $x, $y ), "\n";
```

Checking for Errors: `die` and `warn`

- System calls can fail; examples:

    - Attempt to read a file that doesn't exist

    - Attempt to execute an external program that you do[n't] have permission to execute

- In Perl, use the `die` built in function with the `or` operat[or] terminate (or raise an exception) on error:

    `chdir '/tmp' or die "can't cd to tmp: $!";`

- `die` and `warn` both print a message to STDERR, but `die` will [raise] a fatal exception, `warn` will continue

- If no newline at the end of string, `die` and `warn` print the prog[ram] name and line number where were called

- `$!` holds the value of the last system error message

Parameters — 2

- parameters are passed in one list `@_`.

- If you are passing one parameter, then the builtin function `shift` will conveniently remove the first item from this list, e.g.,

```perl
sub square
{
    my $number = shift;
    return $number * $number;
}
```

Files and Filehandles

- STDIN, STDOUT and STDERR are predefined filehandles

- You can define your own using the `open` built-in function

- Generally use all upper-case letters by convention

- Example: `open` for input:

```
use strict;
open PASSWD, '<', "/etc/passwd"
    or die "unable to open passwd file: $!";
while ( <PASSWD> ) {
    my ( $user ) = split /:/;
    print "$user\n";
}
close PASSWD;
```

Open for Writing

- To create a new file for output, use ">" instead of "<" with the file name.

```
use strict;
open OUT, '>', "data.txt"
    or die "unable to open data.txt: $!";
for ( my $i = 0; $i < 10; ++$i ) {
    print OUT "Time is now ",
        scalar localtime, "\n";
}
close OUT;
```

- Note there is *no comma* after the filehandle in `print`

- To append to a file if it exists, or otherwise create a new file for output, use ">>" instead of ">" with the file name.

Executing External Programs

- Many ways of doing this:

    - `system` built-in function

    - backticks

    - many other ways not covered here.

system

- Example:

```
my @cmd = (
            'useradd',
            '-c', "\"$name\"",
            '-p', $hashed_passwd,
            $id
          );
print "@cmd\n";
system @cmd;
```

- This also works:

```
system "useradd -c \"$name\" -p \"$hashed_passwd\" $id";
```

- difference: second form is usually passed to a command shell (
  as `/bin/sh` or `CMD.EXE`) to execute, whereas the first form is
  cuted directly.

Was `system` Call Successful?

- Check that the return value was zero:

```
if (
    system( "useradd -c \"$name\" -p \"$hashed_passwd\" $id" )
    != 0
  ) {
    print "useradd failed";
    exit;
}
```

- This is usually written in Perl more simply using the built in function **die**, and the **or** operator:

```
system( "useradd -c \"$name\" -p \"$hashed_passwd\" $id" )
        == 0
    or die "useradd failed";
```

Backticks: `'...'` or `qx{...}`

- Perl provides *command substitution*

- Just like in shell programming, where the

- output of the program replaces the code that calls it:

  ```
  print 'ls -l';
  ```

- Note that you can write `qx{...}` instead:

  ```
  print qx{df -h /};
  ```

  - `qx//` is mentioned in slide 29

See the perl summary

- The Perl summary on the subject web site provides... well, a summary!

- Called `perl.pdf`

- Stored in same directory as these notes

Was `system` Call Successful? — 2

- I usually prefer to call `system` like this:

  ```
  my @cmd = (
              'useradd',
              '-c', "\"$name\"",
              '-p', $hashed_passwd,
              $id
            );
  print "@cmd\n";
  system @cmd == 0 or die "Can't execute @cmd";
  ```

Regular Expressions

Regular Expressions are available as part of the programming languages Java, JScript, Visual Basic and VBScript, JavaScript, C++, C#, elisp, Perl, Python, Ruby, PHP, sed, awk, and in many applications, such as editors, grep, egrep.

Regular Expressions help you master your data
— Sales Department.

What is a Regular Expression?

- Powerful.

- Low level description:

  - Describes some text
  - Can use to:
    * Verify a user's input
    * Sift through large amounts of data

- High level description:

  - Allow you to master your data

How to use a Regular Expression

How to make a regular expression as part of you
program

Regular Expressions as a language

- Can consider regular expressions as a language

- Made of two types of characters:

  - *Literal* characters
    * Normal text characters
    * Like words of the program

  - *Metacharacters*
    * The special characters + ? . * ^ $ ( ) [{ | \
    * Act as the grammar that combines with the words according to a set of rules to create and expression that communicates an idea

What do they look like?

- In Perl, a regular expression begins and ends with '/', like
  /abc/

- /abc/ matches the string "abc"

  - Are these literal characters or metacharacters?

- Returns true if matches, so often use as condition in an `if` s
  ment

Example: searching for "Course:"

- Problem: want to print all lines in all input files that contain the string "Course:"

```perl
while ( <> ) {
    my $line = $_;
    if ( $line =~ /Course:/ ) {
        print $line;
    }
}
```

- Or more concisely:

```perl
while ( <> ) {
    print if $_ =~ /Course:/;
}
```

- or even:

```perl
print if /Course:/ while <>;
```

The "match operator" =∼

- If just use /Course:/, this returns true if $_ contains the string "Course:"

- If want to test another string variable $var to see if it contains the regular expression, use

- $var =~ /regular expression/

- Under what condition is this true?

The "match operator" =∼ — 2

```perl
# sets the string to be searched:
$_ = "perl for Win32";

# is 'perl' inside $_?
if ( $_ =~ /perl/ ) { print "Found perl\n" };

# Same as the regex above.
# Don't need the =~ as we are testing $_:
if ( /perl/ )        { print "Found perl\n" };
```

/i — Matching without case sensitivity

```perl
$_ = "perl for Win32";

# this will fail because the case doesn't match:
if ( /PeRl/ )     { print "Found PeRl\n" };

# this will match, because there is an 'er' in 'perl':
if ( /er/ )       { print "Found er\n" };

# this will match, because there is an 'n3' in 'Win32':
if ( /n3/ )       { print "Found n3\n" };

# this will fail because the case doesn't match:
if ( /win32/ )    { print "Found win32\n" };

# This matches because the /i at the end means
# "match without case sensitivity":
if ( /win32/i )   { print "Found win32 (i)\n" };
```

Using !∼ instead of =∼

```
# Looking for a space:
print "Found!\n"  if       / /;


# both these are the same, but reversing the logic with
# unless and !~
print "Found!!\n" unless $_ !~ / /;
print "Found!!\n" unless    !~ / /;
```

Embedding variables in regexps

```
# Create two variables containing
# regular expressions to search for:
my $find = 32;
my $find2 = " for ";


if ( /$find/ )  \{ print "Found '$find'\n" };
if ( /$find2/ ) \{ print "Found '$find2'\n" };
# different way to do the above:
print "Found $find2\n" if /$find2/;
```

- This is the meaning of the "Yes" under "Interpolates" in the table
  on slide 29 on the row for m//

The Metacharacters

The funny characters

What they do

How to use them

Character Classes [...]

```
my @names = ( "Nick", "Albert", "Alex", "Pick" );
foreach my $name ( @names ) {
    if ( $name =~ /[NP]ick/ ) {
        print "$name: Out for a Pick Nick\n";
    else {
        print "$name is not Pick or Nick\n";
    }
}
```

- Square brackets *match one single character*

Examples of use of [...]

- Match a capital letter: `[ABCDEFGHIJKLMNOPQRSTUVWXYZ]`

- Same thing: `[A-Z]`

- Match a vowel: `[aeiou]`

- Match a letter or digit: `[A-Za-z0-9]`

Negated character class: [^...]

- Match any single character that is *not* a letter: `[^A-Za-z]`

- Match any character that is not a space or a tab: `[^ \t]`

Example using [^...]

- This simple program prints only lines that contain characters that are not a space:

```
while ( <> )
{
    print $_ if /[^ ]/;
}
```

- This prints lines that *start with* a character that is not a space:

```
while ( <> ) {
    print if /^[^ ]/;
}
```

- Notice that ^ has two meanings: one inside [...], the other outside.

Matching any character

- The dot matches any character except a newline

- This matches any line with *at least 5* characters before the new

```
print if /...../;
```

Matching the beginning or end

- to match a line that contains *exactly* five characters before newline:

```
print if /^.....$/;
```

- the ^ matches the beginning of the line.

- the $ matches at the end of the line

Shorthand: Common Character Classes

- Since matching a digit is very common, Perl provides \d as a short way of writing [0-9]

- \D matches a non-digit: [^0-9]

- \s matches any whitespace character; shorthand for [\t\n\r\f]

- \S non-whitespace, [^ \t\n\r\f]

- \w word character, [a-zA-Z0-9_]

- \W non-word character, [^a-zA-Z0-9_]

Matching Repetitions: * + ? {n,m}

- To match zero or more:

  - /a*/ will match zero or more letter 'a', so matches "", "aaaa", "qwereqwqwer", or the nothing in front of *anyt*

- to match at least one:

  - /a+/ matches at least one "a"

  - /a?/ matches zero or one "a"

  - /a{3,5}/ matches between 3 and 5 "a"s.

Example using .*

```
$_ = 'Nick Urbanik <nicku@vtc.edu.hk>';
print "found something in <>\bs n" if /<.*>/;

# Find everything between quotes:
$_ = 'He said, "Hi there!", and then "What\'s up?"';
print "quoted!\n" if /"[^"]*"/;
print "too much!\n" if /".*"/;
```

Capturing the match: greediness

- Look at this example:

```
$_ = 'He said, "Hi there!", and then "What\'s up?"';
print "$1\n" if /"([^"]*)"/;
print "$1\n" if /"(.*)"/;
```

- What will each print?

- The first one works; the second one prints:

  ```
  "Hi there!", and then "What's up?
  ```

- Why?

- Because *, ?, +, {m,n} are *greedy*!

- They match as much as they possibly can!

Capturing the Match with (. . .)

- Often want to scan large amounts of data, extracting important items

- Use parentheses and regular expressions

- Silly example of capturing an email address:

  ```
  $_ = 'Nick Urbanik <nicku@vtc.edu.hk>';
  print "found $1 in <>\n" if /<(.*)>/;
  ```

Being Stingy (not Greedy): ?

- Usually greedy matching is what we want, but not always

- How can we match as little as possible?

- Put a ? after the quantifier:

*?	Match 0 or more times
+?	Match 1 or more times
??	Match 0 or 1 time
{n,}?	Match at least n times
{n,m}?	Match at least n, but no more than m times

Being Less Greedy: Example

- We can solve the problem we saw earlier using non-greedy matching:

```
$_ = 'He said, "Hi there!", and then "What\'s up?"';
print "\$1\n" if /"([^"]*)"/;
print "\$1\n" if /"(.*?)"/;
```

- These both work, and match only:

```
Hi there!
```

Capturing the Match: (...)

```
# useradd() is a function defined elsewhere
# that creates a computer account with
# username as first parameter, password as
# the second parameter
while ( <> ) {
    if ( /^(\d{9})\t([A-Z]\d{6}\([\dA]\))/ ) {
        my $student_id = $1;
        my $hk_id = $2;
        useradd( $student_id, $hk_id );
    }
}
```

Sifting through large amounts of data

- Imagine you need to create computing accounts for thousands of students

- As input, you have data of the form:

    - Some heading on the top of each page

    - More headings with other content, including blank lines

    - A tab character separates the columns

    ```
    123456789 H123456(1)
    234567890 I234567(2)
    345678901 J345678(3)
    ...              ...
    987654321 A123456(1)
    ```

The Substitution Operator s///

- Sometimes want to *replace* one string with another (editing)

- Example: want to replace `Nicholas` with `Nick` on input files

    ```
    while ( <> )
    {
        $_ =~ s/Nicholas/Nick/;
        print $_;
    }
    ```

Avoiding leaning toothpicks: /\/\/

- Want to change a filename, edit the directory in the path from, say `/usr/local/bin/filename` to `/usr/bin/filename`

- Could do like this:

  - `s/\/usr\/local\/bin\//\/usr/\bin\//;`

  - but this makes me dizzy!

- We can do this instead:

  - `s!/usr/local/bin/!/usr/bin/!;`

- Can use any character instead of `/` in `s///`

  - For *matches*, can put `m//`, and use any char instead of `/`

  - Can also use parentheses or braces:

  - `s{...}{...}` or `m{...}`

Substitution and the /g modifier

- If an input line contains:

- Nicholas Urbanik read "Nicholas Nickleby"

- then the output is:

- Nick Urbanik read "Nicholas Nickleby"

- How change all the Nicholas in one line?

- Use the `/g` (global) modifier:

```
while ( <> )
{
    $_ =~ s/Nicholas/Nick/g;
    print $_;
}
```

Readable regex: /x Modifier

- Sometimes regular expressions can get long, and need comm inside so others (or you later!) understand

- Use `/x` at the end of `s///x` or `m//x`

- Allows white space, newlines, comments

- See example on slide 9

Special Vars: Input Record Separator

- When I described the `<>` operator, I lied a little

- As `while ( <> ) { ...}` executes, it iterates once per re *not* just once per line.

- The definition of what a record is is given by the special bui variable the *Input Record Separator* `$/`

  - default value is a newline, so by default read one line time

- But useful alternatives are *paragraph mode* and the *whole-file*

Paragraph, Whole-file Modes

- To input in paragraph mode, put this line before you read input:

  ```
  $/ = "";
  ```

- Then when you read input, it will be split at *two or more newlines*

  - You could split the fields at the newlines

- To slurp a whole file into one string, you can do:

  ```
  undef $/;
  $_ = <FILE_HANDLE>; # slurp whole file into $_
  s/\n[ \t]+/ /g;     # fold indented lines
  ```

- See `perldoc -f paragraph`, `perldoc perlvar` and `perldoc -f local` for *important* information on how to localise the change to $/.

`localising` Global Variables

- It is not a good idea to globally change $/, (or even $_)

  - Your program may `use` other modules, and they may be differently if $/ is changed.

  - Best to *localise* the change to $/ (or $_, . . .)

- Example localising whole-file mode:

  ```
  my $content;
  open FH, "foo.txt" or die $!;
  {
      local $/;
      $_ = <FH>;
  }
  close FH;
  ```

- For paragraph mode, put: `local $/ = "";`

One Line Perl Programs

- Called "one liners"

- Just execute on the command line

- See `perldoc perlrun`

- Example:

- `$ perl -pi '.backup' -e 's/Silly/Sensible/g' fileA fileB`

    - edits the files `fileA` and `fileB`

    - makes backups of the original files in `fileA.backup` and `fileB.backup`

    - substitutes all instances of "Silly" and replaces them with "Sensible".

- Useful for editing configuration files in shell scripts, automating tasks

References

- *Learning Perl, 3rd Edition*, Randal L. Schwartz and Tom Phoenix, ISBN 0-596-00132-0, O'Reilly, July 2001.

    - The second edition is fine, too. Don't bother with the first edition, it is too old.

- *Perl Reference Guide*, Johan Vromans, handed out to each one of you, and *will be handed out in the final examination*. Become familiar with it.

- *Perl for System Administration: Managing multi-platform environments with Perl*, David N. Blank-Edelman, ISBN 1-56592-609-9, O'Reilly, July 2000.

- *Perl Cookbook, 2nd Edition*, Tom Christiansen and Nathan Torkington, ISBN 0-596-00313-7, O'Reilly, August 2003

    - The first edition is fine, too.

- Don't forget `perldoc` and all the other documentation on your hard disk.

- *Object Oriented Perl*, Damian Conway, ISBN 1-884777-79-1, Manning, 2000. — A more advanced book for those wanting to build bigger projects in Perl.