

|  |          |
|--|----------|
| Conclusion .....                         | Slide 22 |
| Perl Community                           |          |
| An Overview of Perl .....                | Slide 23 |
| Where do I get Perl? .....               | Slide 24 |
| Where do I get Info about Perl?—1 .....  | Slide 25 |
| Where do I get Info about Perl?—2 .....  | Slide 26 |
| CPAN, PPM: Many Modules .....            | Slide 27 |
| PPM: Perl Package Manager .....          | Slide 28 |
| Mailing Lists: help from experts .....   | Slide 29 |
| How to ask Questions on a List .....     | Slide 30 |
| The Shabang                              |          |
| Where is Perl on my system? .....        | Slide 31 |
| How OS knows it's a Perl program—1 ..... | Slide 32 |
| How OS knows it's a Perl program—2 ..... | Slide 33 |
| Language Overview                        |          |
| Language Overview .....                  | Slide 34 |
| Language Overview — 2 .....              | Slide 35 |
| Data Types                               |          |
| Funny Characters \$, @, % .....          | Slide 36 |
| Arrays .....                             | Slide 37 |
| Array Examples .....                     | Slide 38 |
| More About Arrays .....                  | Slide 39 |
| List Assignment .....                    | Slide 40 |
| Even More About Arrays .....             | Slide 41 |
| Scalar, List Context .....               | Slide 42 |

0-1

|   |          |
|---|----------|
| while and the <> operator — 2 .....         | Slide 64 |
| for loop .....                              | Slide 65 |
| foreach loop .....                          | Slide 66 |
| Iteration                                   |          |
| Iterating over a Hash .....                 | Slide 67 |
| Iterating over a Hash in Sorted Order ..... | Slide 68 |
| Iterating over a Hash in Sorted Order ..... | Slide 69 |
| Other Statements                            |          |
| Exit a Loop Early .....                     | Slide 70 |
| "Backwards" Statements .....                | Slide 71 |
| "Backwards" Statements—Examples .....       | Slide 72 |
| List Operations                             |          |
| Array Operations—push and pop .....         | Slide 73 |
| Array Ops—shift and unshift .....           | Slide 74 |
| split and join .....                        | Slide 75 |
| Subroutines                                 |          |
| Subroutines .....                           | Slide 76 |
| Parameters — 1 .....                        | Slide 77 |
| Parameters — 2 .....                        | Slide 78 |
| Error Handling                              |          |
| Checking for Errors: die and warn .....     | Slide 79 |
| File and Process I/O                        |          |
| Files and Filehandles .....                 | Slide 80 |
| Open for Writing .....                      | Slide 81 |
| Executing External Programs .....           | Slide 82 |

0-3

## Contents

### What is Perl?

|                                  |         |
|----------------------------------|---------|
| What is Perl? .....              | Slide 2 |
| What is Perl? — 2 .....          | Slide 3 |
| Compiled and run each time ..... | Slide 4 |
| Perl is Evolving .....           | Slide 5 |
| Eclectic .....                   | Slide 6 |
| Regular Expressions .....        | Slide 7 |

### Example Problem

|  |          |
|--|----------|
| Why should I learn it? .....                   | Slide 8  |
| The available data .....                       | Slide 9  |
| Sample data for new courses: .....             | Slide 10 |
| Problems .....                                 | Slide 11 |
| Solution in Perl — 1 .....                     | Slide 12 |
| Solution in Perl — 2 .....                     | Slide 13 |
| Solution in Perl — 3 .....                     | Slide 14 |
| But I can use any other language! .....        | Slide 15 |
| Other Solutions may take Longer to Write ..... | Slide 16 |
| The hello world program .....                  | Slide 17 |

### Variables

|                  |          |
|------------------|----------|
| Variables .....  | Slide 18 |
| \$Scalars: ..... | Slide 19 |
| @Array .....     | Slide 20 |
| %Hashes .....    | Slide 21 |

|   |          |
|---|----------|
| Hashes .....                                | Slide 43 |
| Initialising a Hash .....                   | Slide 44 |
| Hash Examples — 1 .....                     | Slide 45 |
| Hash Examples — 2 .....                     | Slide 46 |
| Hash slices .....                           | Slide 47 |
| Another Hash Example .....                  | Slide 48 |
| Hashes are Not Ordered .....                | Slide 49 |
| Good Practice                               |          |
| Discipline—use warnings .....               | Slide 50 |
| use strict and Declaring Variables .....    | Slide 51 |
| Examples of use strict and Variables .....  | Slide 52 |
| Operators, Quoting                          |          |
| Operators and Quoting .....                 | Slide 53 |
| Quoting .....                               | Slide 54 |
| Input, Output                               |          |
| Input and Output .....                      | Slide 55 |
| What is Truth? .....                        | Slide 56 |
| Statements                                  |          |
| Statements for Looping and Conditions ..... | Slide 57 |
| if Statements .....                         | Slide 58 |
| unless Statement .....                      | Slide 59 |
| while loop .....                            | Slide 60 |
| Input with while .....                      | Slide 61 |
| The Special \$_ variable .....              | Slide 62 |
| while and the <> operator .....             | Slide 63 |

0-2

|   |           |
|---|-----------|
| Matching Repetitions: * + ? {n,m} .....     | Slide 107 |
| Example using .* .....                      | Slide 108 |
| Capturing the Match with (...) .....        | Slide 109 |
| Capturing the match: greediness .....       | Slide 110 |
| Being Stingy (not Greedy): ? .....          | Slide 111 |
| Being Less Greedy: Example .....            | Slide 112 |
| Sifting through large amounts of data ..... | Slide 113 |
| Capturing the Match: (...) .....            | Slide 114 |
| The Substitution Operator s/// .....        | Slide 115 |
| Avoiding leaning toothpicks: /\// .....     | Slide 116 |
| Substitution and the /g modifier .....      | Slide 117 |
| Readable regex: /x Modifier .....           | Slide 118 |
| Other Topics                                |           |
| Special Vars: Input Record Separator .....  | Slide 119 |
| Paragraph, Whole-file Modes .....           | Slide 120 |
| localising Global Variables .....           | Slide 121 |
| One Line Perl Programs .....                | Slide 122 |
| References .....                            | Slide 123 |

0-5

1What is Perl?21

|  |           |
|--|-----------|
| system .....                                 | Slide 83  |
| Was system Call Successful? .....            | Slide 84  |
| Was system Call Successful? — 2 .....        | Slide 85  |
| Backticks: `...` or qx{...} .....            | Slide 86  |
| See the perl summary .....                   | Slide 87  |
| Regular Expressions                          |           |
| Regular Expressions .....                    | Slide 88  |
| What is a Regular Expression? .....          | Slide 89  |
| Regular Expressions as a language .....      | Slide 90  |
| How to use a Regular Expression .....        | Slide 91  |
| What do they look like? .....                | Slide 92  |
| Example: searching for "Course:" .....       | Slide 93  |
| The "match operator" =~ .....                | Slide 94  |
| The "match operator" =~ — 2 .....            | Slide 95  |
| /i — Matching without case sensitivity ..... | Slide 96  |
| Using !~ instead of =~ .....                 | Slide 97  |
| Embedding variables in regexps .....         | Slide 98  |
| The Metacharacters .....                     | Slide 99  |
| Character Classes [...] .....                | Slide 100 |
| Examples of use of [...] .....               | Slide 101 |
| Negated character class: [^...] .....        | Slide 102 |
| Example using [^...] .....                   | Slide 103 |
| Shorthand: Common Character Classes .....    | Slide 104 |
| Matching any character .....                 | Slide 105 |
| Matching the beginning or end .....          | Slide 106 |

0-4

1-1

# Perl

## *A language for Systems and Network Administration and Management*

Nick Urbanik

nicku@vtc.edu.hk

A computing department

Copyright Conditions: Open Publication License

(see <http://www.opencontent.org/openpub/>)

## 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 support for Windows in Perl
- Has excellent connection to the operating system
- Has enormous range of modules for thousands of application types

SNM — ver. 1.7

Perl - p. 2/123

## 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 Movie Database
  - Macromedia, Adobe, <http://slashdot.org/>

SNM — ver. 1.7

Perl - p. 4/123

## 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 expressions*
- 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 to do it!” — TIMTOWTDI™

SNM — ver. 1.7

Perl - p. 3/123

## 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

SNM — ver. 1.7

Perl - p. 5/123

## Eclectic

- Borrows ideas from many languages, including:
- C, C++
- Shell
- Lisp
- BASIC
- ... even Fortran
- Many others...

SNM — ver. 1.7

## 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 88

Perl - p. 6/123

SNM — ver. 1.7

Perl - p. 7/123

## 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 and irregular format for now (But Oh, it will be infinitely better in the future! Just wait a year or two...)

1Example Problem81

SNM — ver. 1.7

Perl - p. 8/123

## 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
```

SNM — ver. 1.7

Perl - p. 9/123

SNM — ver. 1.7

Perl - p. 10/123

## 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

```
#!/usr/bin/perl -w
use strict;
my $course;
my $year;
while ( <> )
{
    chomp;
    if ( /^\\s*Course :\\s(\\d+)\\s/ )
    {
        $course = $1;
        undef $year;
        next;
    }
}
```

SNM — ver. 1.7

Perl - p. 11/123

SNM — ver. 1.7

Perl - p. 12/123

## Solution in Perl — 2

```
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
    (
        [A-Z]+     # 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}\d{1}(\d[A-Z])) # HK ID
)*)
)
```

SNM — ver. 1.7

Perl - p. 13/123

## 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

SNM — ver. 1.7

Perl - p. 15/123

## Solution in Perl — 3

```
{
    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+!;
```

SNM — ver. 1.7

Perl - p. 14/123

## 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.

SNM — ver. 1.7

Perl - p. 16/123

# The hello world program

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

17-1

SNM — ver. 1.7

Perl - p. 17/123

## 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 instead of numbers)
- Each type distinguished with a “funny character”

SNM — ver. 1.7

Perl - p. 18/123

## \$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 integer, a floating point number, a character or a string.
  - Note that later you will see that a scalar can also hold a *reference* to another piece of data, which may also be an array or hash.
- Examples:

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

SNM — ver. 1.7

Perl - p. 19/123

## @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 37

SNM — ver. 1.7

Perl - p. 20/123

## % Hashes

- Unfamiliar concept to many of you
- Like an array, but indexed by a string
- A data structure like a database
- See slide 43

SNM — ver. 1.7

Perl - p. 21/123

## Conclusion

- Perl is optimised for text and systems administration programming
- Has great portability
- Is strongly supported by Microsoft
- Has three main built-in data types:
  - Scalar: starts with \$
  - Array: starts with @
  - Hash: starts with %

SNM — ver. 1.7

Perl - p. 22/123

22-1



# An Overview of Perl

A language for Systems and Network  
Administration and Management:

An overview of the language

SNM — ver. 1.7

Perl - p. 23/123

## 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

SNM — ver. 1.7

Perl - p. 25/123

# Where do I get Perl?

- For Windows, go to <http://www.activestate.com>, download the 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

SNM — ver. 1.7

Perl - p. 24/123

## Where do I get Info about Perl?—2

- Web sites:
  - <http://www.perl.com>
  - <http://www.activestate.com>
  - <http://use.perl.org>
- See slide 123 for a list of books.

SNM — ver. 1.7

Perl - p. 26/123

## CPAN, PPM: Many Modules

- A very strong feature of Perl is the community that supports it
- There are tens of thousands of third party modules for many, many purposes:
  - Eg. `Net::LDAP` module supports all LDAP operations, `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 from CPAN, and if so, download it, compile it, test it, and if it passes tests, install it.

SNM — ver. 1.7

Perl - p. 27/123

## 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>

SNM — ver. 1.7

Perl - p. 29/123

## PPM: Perl Package Manager

- For Windows
- Avoids need for a C compiler, other development tools
- Download precompiled modules from ActiveState and other sites, and install them:

```
C:\> ppm install Net::LDAP
```
- See documentation with ActiveState Perl

SNM — ver. 1.7

Perl - p. 28/123

## 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 25

SNM — ver. 1.7

Perl - p. 30/123

## 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 <program>
```
  - OS reads first 2 bytes of program: if they are “#!” then 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 like this:
 

```
$ ./<program>
```

## How OS knows it's a Perl program—2

- **Windows:**
  - OS uses the extension of the file to decide what to do (e.g., `.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 — §36–§52
- compiler warnings, use `strict`; — §50–§52
- operators, quoting — §53–§54
- input and output — §55
- statements: — §57
  - `if...elsif...else` and `unless` statements — §58–§59
  - `while`, `for` and `foreach` loops — §60–§66
    - iterating over arrays and hashes — §66–§69
  - Exit early from a loop with `last`, and `next` — §70
  - “backwards” statements — §71–§72

1Language Overview341

SNM — ver. 1.7

Perl - p. 34/123

## Language Overview — 2

- We also will examine:
  - subroutines, parameters and `return` statement — §76–§78
  - array operations — §73–§75
  - Error reporting: `die` and `warn` — §79
  - Opening files — §80–§81
  - executing external programs — §82–§86
  - regular expressions — §88–§118
  - Special input modes — §119–§121
  - One line Perl programs — §122

35-1

1Data Types361

SNM — ver. 1.7

Perl - p. 35/123

33-1

## 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 called `@while`, and a third called `%while`.

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

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

SNM — ver. 1.7

Perl - p. 36/123

## 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]`, the last as `$array[2]`
- Note that since each element is a scalar, it has the \$ funny character for a scalar variable *value*
- In Perl, *we seldom use an array with an index*—use list processing array operations: `push`, `pop`, `shift`, `unshift`, `split`, `grep`, `map` and iterate over arrays with the `foreach` statement—see slide 66
  - higher level.

SNM — ver. 1.7

Perl - p. 37/123

## Array Examples

- Use the `qw//` “quote words” operator to help initialise arrays — see slide 54

- See slide 66 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" );
```

SNM — ver. 1.7

Perl - p. 38/123

## More About Arrays

- Instead of initialising the array as in slide 38, we can initialise 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
```

SNM — ver. 1.7

Perl - p. 39/123

## List Assignment

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

- 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;
```

SNM — ver. 1.7

Perl - p. 40/123

SNM — ver. 1.7

Perl - p. 41/123

## Even More About Arrays

- How many elements are in the array? See slide 42  

```
print scalar @fruit, "\n"
```
- Does the array contain any data? See slide 59  

```
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";
```

SNM — ver. 1.7

Perl - p. 42/123

## Hashes

- Hashes are probably new to you
- Like an array, but indexed by a string
- Similar idea was implemented in `java.lang.HashMap`
- Perl hashes are easier to use

SNM — ver. 1.7

Perl - p. 43/123

## 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 string on 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 key and which is the value.

SNM — ver. 1.7

Perl - p. 44/123

## Hash Examples — 1

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

```
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:

```
print "$fruit{apple}, $fruit{peach}\n";
prints: crunchy, soft
```

SNM — ver. 1.7

Perl - p. 45/123

## Hash Examples — 2

- How to see if a hash is empty? See 59

```
print "empty\n" unless %fruit;
```
- How to delete a hash element?

```
delete $fruit{coconut};
```
- Hashes are often useful for storing counts (see slides 60–63 for more about while loops):

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

SNM — ver. 1.7

Perl - p. 46/123

## Hash slices

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

```
$score{fred} = 150;
$score{barney} = 100;
$score{dino} = 10;
```
- We could use a *list assignment* (see §40):

```
( $score{fred}, $score{barney}, $score{dino} )
= ( 150, 100, 10 );
... too long. A hash slice makes this easier:
@score{ "fred", "barney", "dino" } = ( 150, 100, 10 );
```
- We can *interpolate* this too (see slides 36 and 54):

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

SNM — ver. 1.7

Perl - p. 47/123

## Another Hash Example

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

```
#!/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 60 for while loop, slide 63 for while ( <> ), slide 66 for the foreach statement, slides 59 and 71 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 67 for how to *iterate* over hash elements

## 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.



## use strict and Declaring Variables

- All programs that are more than a few lines long should have the *pragma* `use strict`;
- This turns on additional checking that all variables are declared, all subroutines are okay, and that references to variables are “hard references” — see `perldoc strict`.
- All variables that you use in your program need to be declared before they are used with either `my` or `our`.
- `my` defines a local variable that exists only in the scope of the current block, or outside of a block, in the file.
  - See `perldoc my`.
- `our` defines a global variable.
  - See `perldoc our`.

SNM — ver. 1.7

Perl - p. 51/123

## 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`

SNM — ver. 1.7

Perl - p. 52/123

## Operators and Quoting

- Perl has all the operators from C (and so Java), in same precedence
- 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 quotes okay in double quotes.
- Documentation in `perldoc perlop`.

SNM — ver. 1.7

Perl - p. 53/123

# Quoting

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

|      |       | Meaning         | Interpolates | Slide    |
|------|-------|-----------------|--------------|----------|
| ' '  | q//   | Literal         | No           | §53, §36 |
| " "  | qq//  | Literal         | Yes          | §53, §36 |
| ` `  | qx//  | Command         | Yes          | §86      |
| ()   | qw//  | quote word list | No           | §38, §70 |
| //   | m//   | Pattern match   | Yes          | §94      |
| s/// | s///  | Substitution    | Yes          | §115     |
| y/// | tr/// | Translation     | No           |          |

- See slide 36 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

1 Statements 571

- We look at the following statements in the language:
  - `if...elsif...else` statements — §58
    - The `unless` statement is similar to the `if` statement — §59
  - `while` loops — §60
    - processing input using `while`
    - The `<>` operator
  - `for` loops — §65
  - `foreach` loops — §66
    - iterating over arrays and hashes with `foreach`, `while` — §66–§69
  - Exit early from a loop with `last`, and `next` — §70
- We will also look at “*backwards statements*” — §71–§72

SNM — ver. 1.7

Perl - p. 57/123

## 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";
}
```

SNM — ver. 1.7

Perl - p. 58/123

## 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

SNM — ver. 1.7

Perl - p. 59/123

56-1

## 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;  
}
```

## 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 line arguments*
  - If you execute it like this:  
angle-brackets.pl  
then you have no *command line arguments* passed to the 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, happen 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 does:

```
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
```

SNM — ver. 1.7

Perl - p. 64/123

## 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...

SNM — ver. 1.7

Perl - p. 65/123

## 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.

SNM — ver. 1.7

Perl - p. 66/123

66-1

Iteration671

## Iterating over a Hash

- Referring to our example hash in slide 43, we can process each element like this:

```
foreach my $key ( keys %hash ) {  
    <process $hash{$key}>  
}
```

- `keys` creates a temporary array of all the keys of the hash
- We then looped through that array with `foreach`.
- More efficient is to use the `each` built in function, which truly iterates through the hash:

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

SNM — ver. 1.7

Perl - p. 67/123

## Iterating over a Hash in Sorted Order

- Did we process the contents of `%hash` in alphabetical order in slide 67?
  - No.
  - So what do we do if we want to print the elements in order?
    - In order of key by alphabet? Numerically?
    - In order of element by alphabet? Numerically?
- Use built in `sort` function
- see `perldoc -f sort`

SNM — ver. 1.7

Perl - p. 68/123

## 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`

SNM — ver. 1.7

Perl - p. 69/123

69-1

## 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?

SNM — ver. 1.7

Perl - p. 70/123

SNM — ver. 1.7

Perl - p. 71/123

## “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;
}
```

SNM — ver. 1.7

Perl - p. 72/123

SNM — ver. 1.7

Perl - p. 71/123

## “Backwards” Statements

- Put an `if`, `while` or `foreach` modifier *after a simple statement*.
- You can put a simple statement (i.e., with no braces), and put one of these afterwards:
  - `if` `EXPR`
  - `unless` `EXPR`
  - `while` `EXPR`
  - `until` `EXPR`
  - `foreach` `EXPR`

72-1

## Array Operations—push and pop

- The documentation for these is in the very loo—ong document `perldoc`, 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`

SNM—ver. 1.7

Perl - p. 73/123

## 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`

SNM—ver. 1.7

Perl - p. 74/123

## split and join

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

- **split** splits a string into an array:

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

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

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

- **join** is the opposite of `split` and joins an array into a string:

```
my $pwwline = join ':', @pwwfields;
```

SNM—ver. 1.7

Perl - p. 75/123

75-1



## Subroutines

- See `perldoc perlsub`

- Syntax:

```
sub <subroutine name>
{
    <statements...>
}
```

SNM — ver. 1.7

Perl - p. 76/123

## Parameters — 1

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

```
#!/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";
```

SNM — ver. 1.7

Perl - p. 77/123

## 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.,

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

SNM — ver. 1.7

Perl - p. 78/123

78-1

1 Error Handling 791

## 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 not have permission to execute
- In Perl, use the `die` built in function with the `or` operator to terminate (or raise an exception) on error:
 

```
chdir '/tmp' or die "can't cd to tmp: $!";
```
- `die` and `warn` both print a message to `STDERR`, but `die` will raise a fatal exception, `warn` will continue
- If no newline at the end of string, `die` and `warn` print the program name and line number where were called
- `$!` holds the value of the last system error message

79-1

SNM — ver. 1.7

Perl - p. 79/123

## 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;
```

SNM — ver. 1.7

Perl - p. 80/123

## 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.

SNM — ver. 1.7

Perl - p. 81/123

# Executing External Programs

- Many ways of doing this:
  - `system` built-in function
  - backticks
  - many other ways not covered here.

SNM — ver. 1.7

Perl - p. 82/123

## 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";
```

SNM — ver. 1.7

Perl - p. 84/123

# `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 (such as `/bin/sh` or `CMD.EXE`) to execute, whereas the first form is executed directly.

SNM — ver. 1.7

Perl - p. 83/123

## 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";
```

SNM — ver. 1.7

Perl - p. 85/123

## 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 54

SNM — ver. 1.7

Perl - p. 86/123

SNM — ver. 1.7

Perl - p. 87/123

## See the perl summary

- The Perl summary on the subject web site provides... well, a good summary!
- Called `perl.pdf`
- Stored in same directory as these notes

## Regular Expressions

Regular Expressions are available as part of the programming languages Java, JScript, Visual Basic and VBScript, JavaScript, C, 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.

1 Regular Expressions 881

SNM — ver. 1.7

Perl - p. 88/123

# 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

SNM — ver. 1.7

Perl - p. 89/123

# How to use a Regular Expression

How to make a regular expression as part of your program

SNM — ver. 1.7

Perl - p. 91/123

# 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

SNM — ver. 1.7

Perl - p. 90/123

# What do they look like?

- In Perl, a regular expression begins and ends with '/', like this: `/abc/`
- `/abc/` matches the string "abc"
  - Are these literal characters or metacharacters?
- Returns true if matches, so often use as condition in an `if` statement

SNM — ver. 1.7

Perl - p. 92/123

## Example: searching for “Course:”

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

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

- Or more concisely:

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

- or even:

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

SNM — ver. 1.7

Perl - p. 93/123

## The “match operator” =~ — 2

```
# 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" };
```

SNM — ver. 1.7

Perl - p. 95/123

## 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?

SNM — ver. 1.7

Perl - p. 94/123

## /i — Matching without case sensitivity

```
$_ = "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" };
```

SNM — ver. 1.7

Perl - p. 96/123

## Using !~ instead of =~

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

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

SNM — ver. 1.7

## The Metacharacters

The funny characters

What they do

How to use them

SNM — ver. 1.7

## 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 54 on the row for m//
```

Perl - p. 97/123

SNM — ver. 1.7

Perl - p. 98/123

## 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*

Perl - p. 99/123

SNM — ver. 1.7

Perl - p. 100/123

## 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]`

SNM — ver. 1.7

Perl - p. 101/123

## 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.

SNM — ver. 1.7

Perl - p. 103/123

## 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]`

SNM — ver. 1.7

Perl - p. 102/123

## 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_]`

SNM — ver. 1.7

Perl - p. 104/123



## Matching any character

- The dot matches any character except a newline
- This matches any line with *at least 5* characters before the newline:

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

SNM — ver. 1.7

Perl - p. 105/123

## Matching Repetitions: \* + ? {n,m}

- To match **zero or more**:
  - `/a*/` will match zero or more letter 'a', so matches "", "a", "aaaa", "qwereqwqwer", or the nothing in front of *anything!*
- 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.

SNM — ver. 1.7

Perl - p. 107/123

## Matching the beginning or end

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

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

- the `^` matches the beginning of the line.
- the `$` matches at the end of the line

SNM — ver. 1.7

Perl - p. 106/123

## 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 /".*"/;
```

SNM — ver. 1.7

Perl - p. 108/123

## 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 /<(.*?)>/;
```

SNM — ver. 1.7

Perl - p. 109/123

## 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

SNM — ver. 1.7

Perl - p. 111/123

## 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!

SNM — ver. 1.7

Perl - p. 110/123

## 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!

SNM — ver. 1.7

Perl - p. 112/123

## 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)
```

SNM — ver. 1.7

Perl - p. 113/123

## 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 $_;
}
```

SNM — ver. 1.7

Perl - p. 115/123

## 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 );
    }
}
```

SNM — ver. 1.7

Perl - p. 114/123

## 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{...}`

SNM — ver. 1.7

Perl - p. 116/123

## 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 $_;  
}
```

SNM — ver. 1.7

Perl - p. 117/123

## Readable regex: /x Modifier

- Sometimes regular expressions can get long, and need comments 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 13

SNM — ver. 1.7

Perl - p. 118/123

## Special Vars: Input Record Separator

- When I described the <> operator, I lied a little
- As `while ( <> ) { ... }` executes, it **iterates once per record**, *not* just **once per line**.
- The definition of what a record is is given by the special built-in variable the **Input Record Separator** `$/`
  - default value is a newline, so by default read one line at a time
- But useful alternatives are **paragraph mode** and the **whole-file mode**

SNM — ver. 1.7

Perl - p. 119/123

## 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 $_`  
`s/\n[ \t]+/ /g; # fold indented lines`
- See `perldoc -f paragraph`, `perldoc perlvar` and `perldoc -f local` for *important* information on how to localise the change to `$/`.

SNM — ver. 1.7

Perl - p. 120/123

## 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

SNM — ver. 1.7

Perl - p. 122/123

## localising Global Variables

- It is not a good idea to globally change `$/`, (or even `$_`)
  - Your program may use other modules, and they may behave 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 $/ = "";`

SNM — ver. 1.7

Perl - p. 121/123

## 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.

SNM — ver. 1.7

Perl - p. 123/123