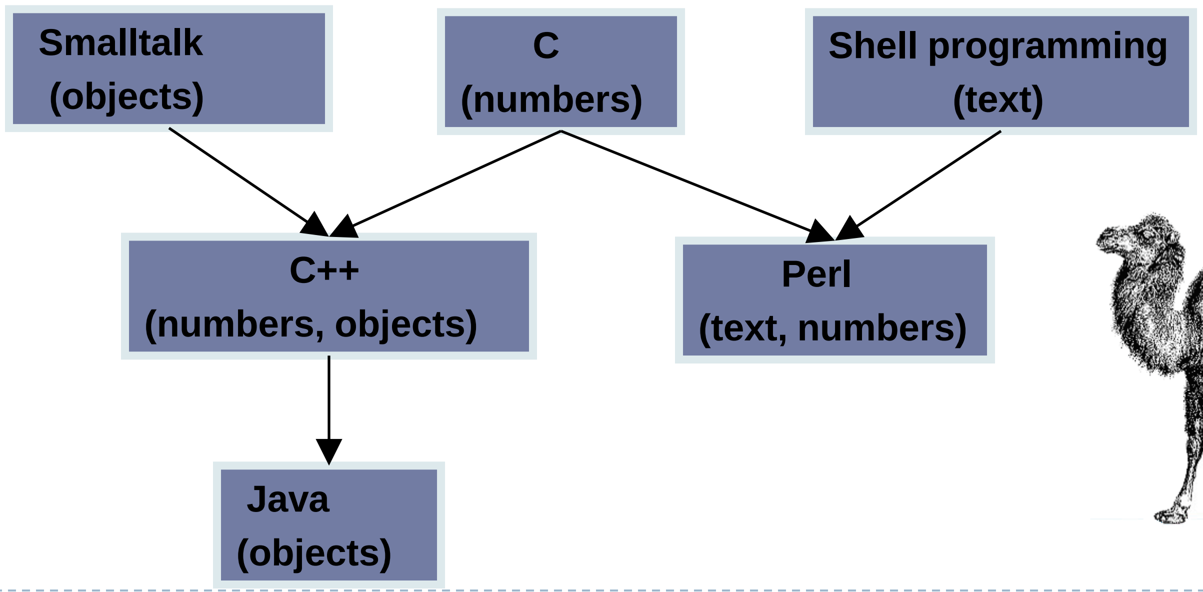
# COMP 2021

## Unix and Script Programming

## Perl Basics

# Perl

- Larry Wall Develops PERL in 1987
- **P**ractical **E**xtraction and **R**eport **L**anguage
  - Wall's original intent was to develop a scripting language more powerful than Unix shell scripting, but not as tedious as C.
  - "Perl is designed to make the easy jobs easy, without making the difficult jobs impossible." -- Larry Wall, Programming Perl



# More about Perl

- Very UNIX-oriented, makes full usage of UNIX power
  - Available on other platforms (including Windows)
  - But isn't always fully implemented there
- An interpreted language, meaning that there is no explicitly separate compilation step
  - more concise and readable way to do many tasks compared to C++ and shell scripts
- Built around regular expressions
  - replaced shell programming as the most popular programming language for text processing and Unix system administration
  - also a popular language for CGI and GUI programming
- Perl does not scale well to large programs
  - Weak subroutines, heavy use of global variables
  - Perl's syntax is not particularly appealing
- Open-source and free language - supported by a helpful international community

# What is perl

- `perl` (lowercase)
  - The language compiler/interpreter program
  - Compiles and interprets source code in single step
  - Accepts many useful command-line arguments for simple "one-line" scripts

```
$ perl -ne 'print' filename
```

- You can run the script directly if you make the script executable:

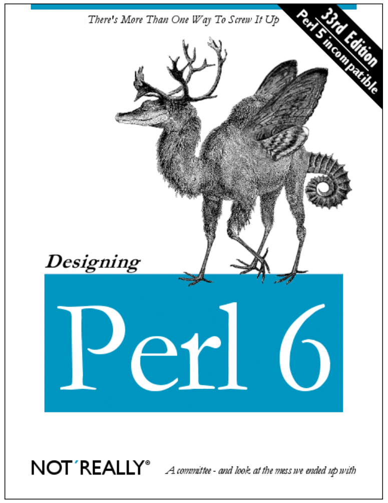
```
$ chmod u+x perlProg.pl
$ perlProg.pl
$
```

## Perl: Hello World

- Here is a "hello world" Perl program

```
$ ls -l helloworld.pl
-rwx------ 1 lixin cs 49 Mar 3  2015 helloworld.pl
$ cat helloworld.pl
#!/usr/local/bin/perl5 -w
# comment lines start with the # character
print "Hello World\n";
$ helloworld.pl
Hello World
```

- Perl code is written in plaintext
  - You can use your favorite text editor to enter code
  - emacs has a perl mode to assist with formatting
- Perl programs are run by perl interpreter


There's More Than One Way To Screw It Up
33rd Edition
Perl 5 incompatible
Designing
Perl 6
NOT REALLY
A committee - and look at the mess we ended up with

## Basic Syntax

- `#!/usr/local/bin/perl5 -w` shebang line with warning message
- Whitespace doesn't matter in Perl (like C++), except for `#!/usr/local/bin/perl5 -w` which must start from column 1 on line 1
- All Perl statements end in a semicolon ; (like C++)
- In Perl, comments begin with `#` (like shell scripts)
  - Everything after the # to the end of the line is ignored.
  - There are no C++-like multiline comments: `/*   */`
- Statements may be combined into blocks with `{}`
- The `print` command sends the string to the screen, and "`\n`" adds a newline.
- You can optionally use parenthesis around the argument

```
print("Hello world\n");
```

## Scalar Variable

- A scalar variable can hold a single value, dynamically assume whatever value is assigned
  - Number: integer (12, 1E+100), real (3.1415926), decimal (15), octal (017), hexadecimal (0xF)
  - String: a single character ("a"), many characters ("A quick brown fox"), unicode ("\x{263A}", UTF-8 format)
- Have a name preceded by a `$` character
  - Must not begin with a digit, case sensitive
  - Special names are reserved (`$_`, `$1`, etc.)
- May be declared in a local scope with a `my` qualifier

## Assigning Scalar Variables

- Scalars are assigned using "="

```
$scalar = expression;
```

- To assign a value to a scalar variable:

```
$number = 25;
$name = "Bill Gates";

$quantity = 6;              # Declare & define
$quantity = "half dozen"; # Now a string
$quantity = 0.5 * 12;      # Numeric again
```

- Unlike shell scripts, use the `$` both when the variable is used and assigned

## Variable Scope

- Variable scope:  enclosing block

```
{
my $email = 'foo@bar.com';
print "$email\n"; # foo@bar.com
}
print $email;
# $email does not exists
```

- Variable Scope: visible everywhere

```
my $lname = "Bar";
print "$lname\n"; # Bar

{
print "$lname\n"; # Bar
$lname = "Other";
print "$lname\n"; # Other
}
print "$lname\n"; # Other
```

## Arithmetic in Perl

- Internally, all numerical scalar values are stored as floats (so you don't have to worry about integer division in Perl like you do in C++).
- Perl supports the usual C++ numerical operations:

```
$a = 25;                          # $a is now 25
$a += 5;                          # $a is now 30
$a *= 3;                          # $a is now 90
$a++;                             # $a is now 91
--$a;                             # $a is now 90
$result = ($a + 2) * 3.4;         # $result is 312.8
$n = 2;                           # Perl has exponentiation
$m = 3;
$result = $n ** $m;
print "$n raised to the $m power is $result\n";
```

- Operator precedence is basically the same as in C++.
- As in C++, you can use parentheses to override precedence, and to clarify the grouping.

## User Input

- Use `<STDIN>` to get input from the user:

```
$ cat read.pl
#!/usr/local/bin/perl5 -w
print "Enter name: ";
$name = <STDIN>;
chomp ($name);
print "How many girlfriends do you have? ";
$number = <STDIN>;
chomp($number);
print "$name has $number girlfriends!\n";
```

## User Input

- `<STDIN>` grabs one line of input, including the newline character. So, after:

  ```
  $name = <STDIN>;
  ```

  if the user typed "Bill Gates[ENTER]", `$name` will contain: "`Bill Gates\n`".
- To delete the newline, the `chomp()` function takes a scalar variable, and removes the trailing newline if present. (If there is no newline at the end, it does nothing.)
- A shortcut to do both operations in one line is:

  ```
  chomp($name = <STDIN>);
  ```

## Quoting Issues

- Characters in single quote ‘’ are interpreted as they are written in the code
- Strings placed between double quotes “” provide interpolation (variables embedded in the string will be replaced by their content), and special escape sequences such as `\t` are replace by a tab
- As with the shell scripts, use a backslash to escape the special meaning
  - `\$, \|,\[, \) ,\*,\^,\/, \\`
- You can use back quote ` ` for command substitution in Perl like in shell scripts

## Example: Quoting

```
$ cat quoting.pl
#!/usr/local/bin/perl5 -w
$user = `whoami`;
chomp($user);
$num = `who | wc -l`;
chomp($num);
print "Hi $user! There are $num users logged on.\n";

print "I have \$5000.\n";
print "It\'s Perl Programming.\n";
print '<-$1500.**>; (update?) [y\n]';
print "\n";
```

- Command substitution will usually include a newline, so use `chomp()`.

## Variable Interpolation

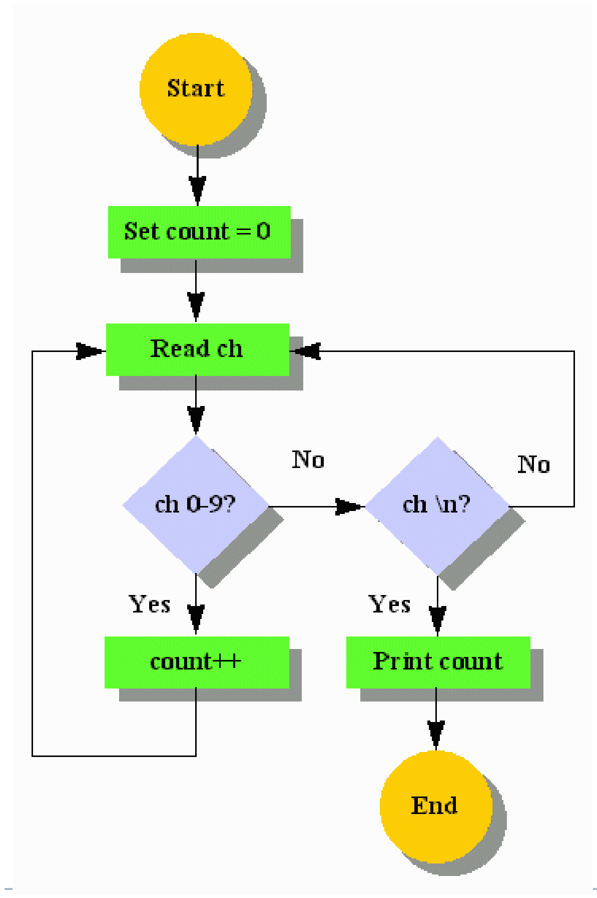
- Putting variables inside double quotes is called variable interpolation.
- The variable name will be the longest possible variable name that makes sense at that part of the string.
- Enclose the variable in a pair of curly braces if needed to override this.

```
$ cat interpolation.pl
#!/usr/local/bin/perl5 -w
$comp = "difficult";
$comp2021 = "fun";

print "COMP subject is $comp?\n";
print "But COMP2021 is $comp2021!\n";
print "COMP subject is ${comp}2021\n";
print "COMP "."2021 ". "is "."$comp2021.\n";
$ interpolation.pl
COMP subject is difficult?
But COMP2021 is fun!
COMP subject is difficult2021
COMP 2021 is fun.
```

## Control Flow

- Perl has several control flow statements:
  - `if`
  - `while`
  - `for`
  - *`unless`*
  - *`until`*
  - *`do while`*
  - *`do until`*
  - *`foreach`*



## `if`

- The Perl `if` statement works almost the same as in C++:

```
#!/usr/local/bin/perl5 -w
$user = `whoami`;
chomp($user);
if($user eq "cindy"){
        print "Hi Cindy!\n";
}
```

if you can dream and not make dreams your master
if you can think and not make thoughts your aim
if you can meet with Triumph and Disaster
and treat those two impostors just the same

- The curly braces `{ }` are always required in Perl (even if only one statement inside, unlike C++). This avoids the "dangling else" problem.

## if else

- The `if else` statement is similar:

```
#!/usr/local/bin/perl5 -w
$user = `whoami`;
chomp($user);
if ($user eq “cindy") {
      print "Hi Cindy!\n";
} else {
       print "Hi $user!\n";
}
```

## if elsif else

- You can also handle a list of cases:

```
#!/usr/local/bin/perl5 -w
$users = `who | wc -l`;
chomp($users);
if ($users > 4){
      print "Heavy load!\n";
}
elsif ($users > 1){
       print "Medium load\n";
}
else {
       print "Just me!\n";
}
```

## Relational Operators

- Perl's numeric and string comparison operators:

| Comparison | Numeric | String |
|---|---|---|
| Equal | `==` | `eq` |
| Not equal | `!=` | `ne` |
| Less than | `<` | `lt` |
| Greater than | `>` | `gt` |
| Less than or equal to | `<=` | `le` |
| Greater than or equal to | `>=` | `ge` |

## Truth in Perl

- Truth is flexible in Perl:
  - Expressions that evaluate to false

```
    0                   # traditional false value
    ""                  # the null string
  "0"                   # only non-zero length false string
```

  - Some examples of truth:

```
     1                  # traditional true value
     684                # non-zero numerical values are true
     " "                # whitespace is true
    "hello"             # strings are true
    "00"                # a string
```

# Example: Comparison

| | |
|---|---|
| `12.0 == 12` | TRUE |
| `"12.0" == 12` | TRUE (== operator of Perl converts the string to a number) |
| `"12.0" eq 12` | FALSE |
| `2 < 3` | TRUE |
| `2 lt 3` | TRUE (2 is before 3 in the ASCII table) |
| `12 > 3` | TRUE |
| `12 gt 3` | FALSE ! (look out, might not be obvious at first) |
| `"foo" == " "` | TRUE ! (get warnings if you used the "warnings" pragmata) |
| `"foo" eq " "` | FALSE |
| `"foo" == "bar"` | TRUE (warnings if you used the "warnings" pragmata) |
| `"foo" eq "bar"` | FALSE |

## And, Or, Not

- `1` represents true, and `0` false (as in C++).
- You can also combine and negate expressions with logical and (`&&`), logical or (`||`), and not (`!`) just like in C++:

```
#!/usr/local/bin/perl5 -w
chomp($user = `whoami`);
chomp($nme = `who | grep $user | wc -l`);
chomp($nusers = `who | wc -l`);
if($nusers - $nme && $user ne “lixin"){
     print "Someone else is logged in!\n";
}
else{
     print “I feel lonely!\n";
}
```

## while

- The `while` statement loops indefinitely, while the condition is true, such as a user-controlled condition:

```
#!/usr/local/bin/perl5 -w
$resp = "no";
while($resp ne "yes"){
    print "Wakeup [yes/no]? ";
    chomp($resp = <STDIN>);
}
```

## for

- for can be used as in C++ to do incrementing loops:

```
$ cat factorial.pl
#!/usr/local/bin/perl5 -w
print "Enter number: ";
chomp($n = <STDIN>);

$fac = 1;
for($i=1; $i<=$n; $i++){
     $fac *= $i;
}
print "The factorial of $n is $fac\n";
$ fac
Enter number: 5
The factorial of 5 is 120
```

## `last`

- The `last` command works like the C++ `break` command, breaking out of the innermost loop :

```
$ cat wakeup.pl
#!/usr/local/bin/perl5 -w
while(1){
     print "Wakeup [yes/no]? ";
     chomp($resp = <STDIN>);
     if($resp eq "yes"){
           last;
     }
}
$ wakeup.pl
Wakeup [yes/no]? no
Wakeup [yes/no]? y
Wakeup [yes/no]? yes
```

## Exercise: Number Guess Game

- Computer generates a random number 1-100 and you need to guess it
- Random number
  - `$z = int rand 6; # generate a random integer in between 0 to 5`