What is Perl?

- Practical Extraction and Report Language
- Scripting language created by Larry Wall in the mid-80s
- Functionality and speed somewhere between low-level languages (like C) and high-level ones (like shell)
- Influence from awk, sed, and C Shell
- Easy to write (after you learn it), but sometimes hard to read
- Widely used in CGI scripting

A Simple Perl Script

```perl
#!/usr/bin/perl -w
print "Hello, world!\n";
```

Another Perl Script

```perl
$s=0.20^
087{(+e, a||0-1)})$e Vec4{hsu, (1!b52) (n)
|/1%2s}, |0n|-9e)le [0u]2(5", [h-3h 5
hc?e8?1iou, t([1e885e0(19)g2, $31)+.5(4"n); (95, 1b2x_.i.
we7]/7[1e885b5])|4, a||()]|85, jx, aln5a+u), x, x, x, x, x, x, x, x
1[05, 0xent.c] [1[1e885b5|d|d]]+2, 1[1e885b5]m{e[4e885b2(0)]; a[1", *7], [1e885b2/0]/20]|21, 0(15, 1b54:5
97r_7)|$2|5[4e885b2(0)]1|0]|0(1d3
Or1, 5": open(wt;2);. =07; for($e=2; $e<666; $e+=+5)$e
(1(1)); push@S, 5;+n=5d0$e); Xor[++X, 1
S85]; (Op=7, 0); +08|Sphsubehs(Sa, 5|1, 3)+1]1]|w{152;
```

Data Types

- Basic types: scalar, lists, hashes
- Support OO programming and user-defined types

What Type?

- Type of variable determined by special leading character
  - `$foo` scalar
  - `@foo` list
  - `%foo` hash
  - `&foo` function

- Data types have separate name spaces
Scalars

- Can be numbers
  
  \$num = 100;
  \$num = 233.45;
  \$num = -1.3e38;

- Can be strings
  
  \$str = "unix tools";
  \$str = "Who\'s there??";
  \$str = "good evening\n";
  \$str = "one	two";

- Backslash escapes and variable names are interpreted inside double quotes

Special Scalar Variables

<table>
<thead>
<tr>
<th></th>
<th>Name of script</th>
</tr>
</thead>
<tbody>
<tr>
<td>$0</td>
<td></td>
</tr>
<tr>
<td>$_</td>
<td>Default variable</td>
</tr>
<tr>
<td>$$</td>
<td>Current PID</td>
</tr>
<tr>
<td>$#</td>
<td>Status of last pipe or system call</td>
</tr>
<tr>
<td>$!</td>
<td>System error message</td>
</tr>
<tr>
<td>$/</td>
<td>Input record separator</td>
</tr>
<tr>
<td>$</td>
<td>Input record number</td>
</tr>
<tr>
<td>undefined</td>
<td>Acts like 0 or empty string</td>
</tr>
</tbody>
</table>

Operators

- Numeric: + * / % **
- String concatenation: .
  
  \$state = "New" . "York";  # "NewYork"

- String repetition: x
  
  print "bla" x 3;  # blablabla

- Binary assignments:
  
  \$val = 2; \$val *= 3;  # $val is 6
  \$state .= "City";  # "NewYorkCity"

Comparison Operators

<table>
<thead>
<tr>
<th>Comparison</th>
<th>Numeric</th>
<th>String</th>
</tr>
</thead>
<tbody>
<tr>
<td>==</td>
<td>eq</td>
<td></td>
</tr>
<tr>
<td>!=</td>
<td>ne</td>
<td></td>
</tr>
<tr>
<td>&gt;</td>
<td>gt</td>
<td></td>
</tr>
<tr>
<td>&lt;</td>
<td>lt</td>
<td></td>
</tr>
<tr>
<td>&lt;=</td>
<td>le</td>
<td></td>
</tr>
<tr>
<td>&gt;=</td>
<td>ge</td>
<td></td>
</tr>
</tbody>
</table>

Boolean “Values”

```perl
if ($ostype eq "unix") { ... }
if ($val) { ... }
```

- No boolean data type
- undefined is false
- 0 is false; Non-zero numbers are true
- ‘’ and ‘0’ are false; other strings are true
- The unary not (!) negates the boolean value

Undefined and defined

```perl
$f = 1;
while ($n < 10) { 
  # $n is undefined at 1st iteration
  $f *= ++$n;
}
```

- Use defined to check if a value is undefined
  ```perl
  if (defined($val)) { ... }
  ```
Lists and Arrays

- List: ordered collection of scalars
- Array: Variable containing a list
- Each element is a scalar variable
- Indices are integers starting at 0

Array/List Assignment

@teams=(“Knicks”, “Nets”, “Lakers”);
print $teams[0];  # print Knicks
$teams[3]=“Celtics”;  # add new elt
@foo = ();  # empty list
@nums = (1..100);  # list of 1-100
@arr = ($x, $y*6);
($a, $b) = (“apple”, “orange”);
($a, $b) = ($b, $a);  # swap $a $b
@arr1 = @arr2;

Array/List Assignment

More About Arrays and Lists

- Quoted words - qw
  @planets = qw/ earth mars jupiter /;
  @planets = qw{ earth mars jupiter };
  Last element’s index: $#planets
  – Not the same as number of elements in array!
  Last element: $planets[-1]

Scalar and List Context

- Array interpolated as string:
  print “My favorite colors are @colors
  # Prints My favorite colors are red green blue
- Array in scalar context returns the number of elements in the list
  $num = @colors + 5;  # $num gets 8
- Scalar expression in list context
  @num = 88;  # a one-element list (88)

pop and push

- push and pop: arrays used as stacks
  @colors = qw# red green blue #;
  push(@colors, “yellow”);  # same as
  @colors = (@colors, “yellow”);
  push @colors, @more_colors;
- pop removes last element of array and returns it
  $lastcolor = pop(@colors);

shift and unshift

- shift and unshift: similar to push and pop on the “left” side of an array
  @colors = qw# red green blue #;
  unshift @colors, “orange”;
  First element is now “orange”
- shift removes element from beginning
  @c = shift(@colors);  # @c gets “orange”
**sort and reverse**

- `reverse` returns list with elements in reverse order
  
  ```perl
  @list1 = qw NY NJ CT #;  # (CT,NJ, NY)
  @list2 = reverse(@list1); # (CT, NJ, NY)
  ```

- `sort` returns list with elements in ASCII order
  
  ```perl
  @day = qw /tues wed thurs/;  # (thurs, tues, wed)
  @sorted = sort(@day); # (thurs, tues, wed)
  ```

- `reverse` and `sort` do not modify their arguments

**Iterate over a List**

- `foreach` loops through a list of values
  
  ```perl
  @teams = qw Knicks Nets Lakers #;
  foreach $team (@teams)
  {
    print "$team win
  }
  ```

- Value of control variable restored at end of loop

- `$_` is the default

```perl
foreach (@teams) 
{ 
$_ = "win
; print;
}
```

**Hashes**

- Associative arrays - indexed by strings (keys)
  
  ```perl
  $cap{"Hawaii"} = "Honolulu";
  $cap = ( "New York", "Albany", "New Jersey", "Trenton", "Delaware", "Dover" );
  ```

- Can use => (big arrow or comma arrow) in place of `if` (comma)

  ```perl
  %cap = ( "New York" => "Albany", "New Jersey" => "Trenton", "Delaware" => "Dover" );
  ```

**Hash Functions**

- `keys` returns a list of keys
  
  ```perl
  @state = keys %cap;
  ```

- `values` returns a list of values
  
  ```perl
  @city = values %cap;
  ```

- Use `each` to iterate over all (key, value) pairs while `if`

  ```perl
  while ( ($state, $city) = each %cap )
  {
    print "Capital of $state is $city\n"
  }
  ```

**Hash Element Access**

- `Hash{key}`

  ```perl
  print $cap{"New York"};
  print $cap{"New " . "York"};
  ```

- Unwinding the hash

  ```perl
  @cap_arr = %cap;
  ```

- Assigning one hash to another

  ```perl
  %cap2 = %cap;
  %cap_of = reverse %cap;
  print $cap_of{"Trenton"}; # New Jersey
  ```

**Hash Element Interpolation**

- Unlike a list, entire hash cannot be interpolated

  ```perl
  print "%cap\n";
  ```

- Prints `%cap` followed by a newline

- Individual elements can

  ```perl
  foreach $state (sort keys %cap) 
  { 
    print "Capital of $state is $cap($state)\n"
  }
  ```
More Hash Functions

- exists checks if a hash element has ever been initialized
  - Can be used for array elements
  - A hash or array element can only be defined if it exists
- delete removes a key from the hash

```perl
$cap{"New York"};
```

Merging Hashes

- Method 1: Treat them as lists
  - Can be used for array elements
  - A hash or array element can only be defined if it exists
- Method 2 (save memory): Build a new hash by looping over all elements
  - A hash or array element can only be defined if it exists

```perl
%h3 = (%h1, %h2);
```

Subroutines

- sub myfunc { ...
  $name="Jane";
  ...
  sub print_hello {
    print "Hello $name\n"; # global $name
  }
  print_hello; # print "Hello Jane"
  print_hello(); # print "Hello Jane"
```

Arguments

- Parameters are assigned to the special array @_
- Individual parameter can be accessed as $_[0], $_[1], ...
  - Inside func:
    - $_[0] is $x
    - $_[1] is $y[0]
    - $_[2] is $y[1], etc.
- Scalars in @_ are implicit aliases (not copies) of the ones passed — changing values of $_[0], etc. changes the original variables
  ```perl
  sub sum {
    my $x; # private variable $x
    foreach (@_) { # iterate over params
      $x += $_[0];
    }
    return $x;
  }
  $n = &sum(3, 10, 22); # $n gets 35
  ```

More on Parameter Passing

- Any number of scalars, lists, and hashes can be passed to a subroutine
- Lists and hashes are "flattened"
  ```perl
  func($x, @y, %z);
  ```
- Lists and hashes are "flattened"
  ```perl
  func($x, @y, %z);
  ```
- Scalars in @_ are implicit aliases (not copies) of the ones passed — changing values of $_[0], etc. changes the original variables
  ```perl
  sub myfunc {
    my $x = 1;
    $x + 2; # returns 3
  }
  sub myfunc {
    my $x = 1;
    $x + 2; # returns 3
  }
  ```

Return Values

- The return value of a subroutine is the last expression evaluated, or the value returned by the return operator
  ```perl
  sub myfunc {
    my $x = 1;
    $x + 2; # returns 3
  }
  sub myfunc {
    my $x = 1;
    $x + 2; # returns 3
  }
  ```
- Can also return a list: return @somelist;
- If return is used without an expression (failure), undef or [] is returned depending on context
Lexical Variables

- Variables can be scoped to the enclosing block with the `my` operator
  ```perl
  sub myfunc {
      my $x;
      my($a, $b) = @_;  # copy params
  }
  ```
- Can be used in any block, such as if block or while block
  - Without enclosing block, the scope is the source file

use strict

- The `use strict` pragma enforces some good programming rules
  - All new variables need to be declared with `my`
  ```perl
  #!/usr/bin/perl -w
  use strict;
  $n = 1;  # c-- perl will complain
  ```

Another Subroutine Example

```perl
$nums = (1, 2, 3);
@res = dec_by_one(@nums);  # @res=(1, 2, 3)
```

Reading from STDIN

- STDIN is the built-in filehandle to the std input
- Use the line input operator around a file handle to read from it
  ```perl
  while (<STDIN>) {
      chomp;
      print "Line \$_. --> \$_\n";
  }
  ```
- `chomp` removes trailing string that corresponds to the value of `$/` (usually the newline character)

Reading from STDIN example

```perl
while (<STDIN>) {
    chomp;
    print "Line \$_. --> \$_\n";
}
```

<>

- Diamond operator `< >` helps Perl programs behave like standard Unix utilities (cut, sed, ...)
- Lines are read from list of files given as command line arguments (@ARGV), otherwise from stdin
  ```perl
  while (<>) {  
      chomp;
      print "Line \$_. from $ARGV is \$_\n";
  }
  ```
- `/myprog file1 file2`
  - Read from `file1`, then `file2`, then standard input
- `$ARGV` is the current filename
**Filehandles**

- Use `open` to open a file for reading/writing
  - `open LOG, "syslog";`  
    # read
  - `open LOG, "<syslog";`  
    # read
  - `open LOG, ">syslog";`  
    # write
  - `open LOG, ">>syslog";`  
    # append
- When you're done with a filehandle, close it
  - `close LOG;`

**Errors**

- When a fatal error is encountered, use `die` to print out error message and exit program
  - `die "Something bad happened\n" if ...;
- Always check return value of `open`
  - `open LOG, ">>syslog"`  
    or `die "Cannot open log: $!";`
- For non-fatal errors, use `warn` instead
  - `warn "Temperature is below 0!"`  
    if `$temp < 0;`

**Reading from a File**

```perl
open MSG, "/var/log/messages"
  or die "Cannot open messages: $!\n";
while (<MSG>) {
  chomp;
  # do something with $_
}
close MSG;
```

**Reading Whole File**

- In scalar context, `<FH>` reads the next line
  - `$line = <LOG>;`
- In list context, `<FH>` read all remaining lines
  - `@lines = <LOG>;`
- Undefine `$/` to read the rest of file as a string
  - `undef $/;
    $all_lines = <LOG>;`

**Writing to a File**

```perl
open LOG, ">/tmp/log"
  or die "Cannot create log: $!";
print LOG "Some log messages...\n"
printf LOG "$d entries processed:\n", $num;
close LOG;
```

**File Tests**

```perl
die "The file $filename is not readable"  
  if ! -r $filename;
warn "The file $filename is not owned by you unless -o $filename;"
print "This file is old" if -M $filename > 365;
```
### File Tests

<table>
<thead>
<tr>
<th>Test</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-r</td>
<td>File or directory is readable</td>
</tr>
<tr>
<td>-w</td>
<td>File or directory is writable</td>
</tr>
<tr>
<td>-x</td>
<td>File or directory is executable</td>
</tr>
<tr>
<td>-o</td>
<td>File or directory is owned by this user</td>
</tr>
<tr>
<td>-e</td>
<td>File or directory exists</td>
</tr>
<tr>
<td>-z</td>
<td>File exists and has zero size</td>
</tr>
<tr>
<td>-s</td>
<td>File or directory exists and has nonzero size (value in bytes)</td>
</tr>
</tbody>
</table>

### File Tests

<table>
<thead>
<tr>
<th>Test</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-f</td>
<td>Entry if a plain file exists and has zero size</td>
</tr>
<tr>
<td>-d</td>
<td>Entry is a directory</td>
</tr>
<tr>
<td>-l</td>
<td>Entry is a symbolic link</td>
</tr>
<tr>
<td>-M</td>
<td>Modification age (in days)</td>
</tr>
<tr>
<td>-A</td>
<td>Access age (in days)</td>
</tr>
</tbody>
</table>

* $._$ is the default operand

### Manipulating Files and Dirs

- `unlink` removes files
  `unlink "file1", "file2"`
  or warn "failed to remove file: $!";
- `rename` renames a file
  `rename "file1", "file2"`;
- `link` creates a new (hard) link
  `link "file1", "file2"`
  or warn "can't create link: $!";
- `symlink` creates a soft link
  `link "file1", "file2"` or warn " _ _ ";

### Manipulating Files and Dirs

- `mkdir` creates directory
  `mkdir "mydir", 0755`
  or warn "Cannot create mydir: $!";
- `rmdir` removes empty directories
  `rmdir "dir1", "dir2", "dir3"`;
- `chmod` modifies permissions on file or directory
  `chmod 0600, "file1", "file2"`;

### if - elsif - else

```perl
if { $x > 0 } {
  print "x is positive\n";
} elsif { $x < 0 } {
  print "x is negative\n";
} else {
  print "x is zero\n";
}
```

### unless

```perl
unless ($x < 0) {
  print "$x is non-negative\n";
}
```

```
unlink $file unless -A $file < 100;
```
while and until

```php
while ($x < 100) {
    $y += $x++;
}
• until is like the opposite of while
until ($x >= 100) {
    $y += $x++;
}
```

for

```php
• for (init; test; incr) { ... }

# sum of squares of 1 to 5
for ($i = 1; $i <= 5; $i++) {
    $sum += $i*$i;
}
```

next

```php
• next skips the remaining of the current
iteration (like continue in C)

# only print non-blank lines
while (<>) {
    if ( $_ eq "\n") { next; }
    else { print; }
}
```

last

```php
• last exits loop immediately (like break in C)

# print up to first blank line
while (<>){
    if ( $_ eq "\n") { last; }
    else { print; }
}
```

Logical AND/OR

```php
• Logical AND : &&
  if (($x > 0) && ($x < 10)) { ... }

• Logical OR : ||
  if ($x < 0) || ($x > 0)) { ... }

• Both are short-circuit — second expression
  evaluated only if necessary
```

Ternary Operator

```php
• Same as the ternary operator (?:) in C
  expr1 ? expr2 : expr3

• Like if-then-else: If expr1 is true, expr2 is
  used; otherwise expr3 is used

$weather=($temp>50)?"warm":"cold";
```
Regular Expressions

• Use EREs (egrep style)
• Plus the following character classes
  - \w “word” characters: [A-Za-z0-9_]
  - \d digits: [0-9]
  - \s whitespaces: [\f\t\n\r ]
  - \b word boundaries
  - \W, \D, \S, \B are complements of the corresponding classes above
• Can use \ to denote a tab

Backreferences

• Support backreferences
• Subexpressions are referred to using \1, \2, etc. in the RE and \$1, \$2, etc. outside RE

```
if /^(?this (red|blue|green) (bat|ball) is \1/) {
  ($color, $object) = ($1, $2);
}
```

Matching

• Pattern match operator: /RE/ is shortcut of m/RE/
  – Returns true if there is a match
  – Match against $_
  – Can also use m(RE), m<RE>, m!RE, etc.
  if (/^\usr/\local/) { ... }
if (m\usr/\local/) { ... }
• Case-insensitive match
  if (/\new york/i) { ... }

Matching_cont.

• To match an RE against something other than $_, use the binding operator =~
  if ($s =~ /\bblah/i) {
    print "Found blah!"
  }
• !~ negates the match
  while (<STDIN> !~ /^#/) { ... }
• Variables are interpolated inside REs
  if (/^$word/) { ... }

Substitutions

• Sed-like search and replace with s///
  s/\red/\blue/;
  $x =~ s/\w+$/$\2/;
• m/// does not modify variable; s/// does
• Global replacement with /g
  s/(\.)/\1/$1/g;
• Transliteration operator: tr/// or y///
  tr/A-Z/a-z/;

RE Functions

• split string using RE (whitespace by default)
  @fields = split "/\", "ab:code:ef";
  # gets ("","ab","code","ef")
• join strings into one
  @str = join ",", @fields;  # gets "ab-cde-f"
• grep something from a list
  – Similar to UNIX grep, but not limited to using RE
  – Modifying elements in returned list actually modifies the elements in the original list
  @selected = grep {!/\#/, @code};
  @matched = grep ( $s>100 && $s<150 ) @nums;
Running Another Program

- Use the `system` function to run an external program
- With one argument, the shell is used to run the command
  - Convenient when redirection is needed
    ```
    $status = system("cmd1 args > file");
    ```
- To avoid the shell, pass `system` a list
  ```
  $status = system($prog, @args);  
die "$prog exited abnormally: $?" unless $status == 0;
  ```

Capturing Output

- If output from another program needs to be collected, use the backticks
  ```
  my $files = `ls *.c`;  
  ```
  - Collect all output lines into a single string
    ```
    my @files = `ls *.c`;  
    ```
  - Each element is an output line
  - The shell is invoked to run the command

Environment Variables

- Environment variables are stored in the special hash `%ENV`
  ```
  %ENV{‘PATH’} = 
  "/usr/local/bin:$ENV{‘PATH’}";
  ```

Example: Word Frequency

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

use strict;
my (@words, %count, $word);

chomp(@words = <STDIN>);  
for $word (@words) {
  $count{$word}++;
}

for $word (keys %count) {
  print "$word was seen $count{$word} times.
";
}
```

Good Ways to Learn Perl

- `a2p` - Translates an awk program to Perl
- `s2p` - Translates a sed script to Perl
- `perldoc` - Online Perl documentation
  ```
  0 perldoc perldoc  -- perldoc man page  
  0 perldoc perlintro -- Perl introduction  
  0 perldoc -f sort  -- Perl sort function man page  
  0 perldoc CGI  -- CGI module man page
  ```

Modules

- Perl modules are libraries of reusable code with specific functionalities
- Standard modules are distributed with Perl, others can be obtained from `CPAN`
- Include modules in your program with use, e.g. use `CGI` incorporates the CGI module
- Each module has its own namespace
Forms

- HTML forms are used to collect user input
- Data sent via HTTP request
- Server launches CGI script to process data

```html
<form method=POST action="http://www.cs.nyu.edu/~unixtool/cgi-bin/search.cgi">
Enter your query: <input type=text name=Search> <input type=submit>
</form>
```

Input Types

- Text Field
  ```html
  <input type=text name=zipcode>
  ```
- Radio Buttons
  ```html
  <input type=radio name=size value="S"> Small
  <input type=radio name=size value="M"> Medium
  <input type=radio name=size value="L"> Large
  ```
- Checkboxes
  ```html
  <input type=checkbox name=extras value="lettuce"> Lettuce
  <input type=checkbox name=extras value="tomato"> Tomato
  ```
- Text Area
  ```html
  <textarea name=address cols=50 rows=4>…
  </textarea>
  ```

Submit Button

- Submits the form for processing by the CGI script specified in the form tag
  ```html
  <input type=submit value="Submit Order"> 
  ```

HTTP Methods

- Determine how form data are sent to web server
- Two methods:
  - GET
    - Form variables stored in URL
  - POST
    - Form variables sent as content of HTTP request
Encoding Form Values

- Browser sends form variable as name-value pairs
  - `name1=value1&name2=value2&name3=value3`
- Names are defined in form elements
  - `<input type=text name=ssn maxlength=9>`
- Special characters are replaced with `%##` (2-digit hex number), spaces replaced with `+`
  - e.g. “11/8 Wed” is encoded as “11%2F8+Wed”

HTTP GET/POST examples

**GET:**
- `GET /cgi-bin/myscript.pl?name=Bill%20Gates&company=Microsoft HTTP/1.1 HOST: www.cs.nyu.edu`

**POST:**
- `POST /cgi-bin/myscript.pl HTTP/1.1 HOST: www.cs.nyu.edu
  ...
  name=Bill%20Gates&company=Microsoft`

GET or POST?

- GET method is useful for
  - Retrieving information, e.g. from a database
  - Embedding data in URL without form element
- POST method should be used for forms with
  - Many fields or long fields
  - Sensitive information
  - Data for updating database
- GET requests may be cached by clients browsers or proxies, but not POST requests

Parsing Form Input

- Method stored in `HTTP_METHOD`
- GET: Data encoded into `QUERY_STRING`
- POST: Data in standard input (from body of request)
- Most scripts parse input into an associative array
  - You can parse it yourself
  - Or use available libraries (better)

CGI Script: Example

```html
<html>
<title>Anonymous Comment Submission</title>
<body>
Please enter your comment below which will be sent anonymously to <tt>kornj@cs.nyu.edu</tt>.
If you want to be extra cautious, access this page through `<a href="http://www.anonymizer.com">Anonymizer</a>`.

<form action=/cgi-bin/comment.pl method=post>
<textarea name=comment rows=20 cols=80></textarea>
<input type=submit value="Submit Comment">
</form>
</body>
</html>
```

Part 1: HTML Form

```html
<html>
<title>Anonymous Comment Submission</title>
<body>
Please enter your comment below which will be sent anonymously to <tt>kornj@cs.nyu.edu</tt>.
If you want to be extra cautious, access this page through `<a href="http://www.anonymizer.com">Anonymizer</a>`.

<form action=/cgi-bin/comment.pl method=post>
<textarea name=comment rows=20 cols=80>
</textarea>
<input type=submit value="Submit Comment">
</form>
</body>
</html>
```
Part 2: CGI Script (ksh)

```bash
#!/home/unixtool/bin/ksh
. cgi-lib.ksh # Read special functions to help parse
ReadParse
PrintHeader
print -r -- "${Cgi.comment}" | /bin/mailx -s "COMMENT" kornj
print "<H2>You submitted the comment</H2>"
print "<pre>
print -r -- "${Cgi.comment}" 
print "</pre>
```

Perl CGI Module

- Interface for parsing and interpreting query strings passed to CGI scripts
- Methods for creating generating HTML
- Methods to handle errors in CGI scripts
- Two interfaces: procedural and OO
  - Ask for the procedural interface:
    ```bash
    use CGI qw(:standard);
    ```

A Perl CGI Script

```perl
#!/usr/bin/perl -w
use strict;
use CGI qw(:standard);
my $bday = param("birthday");
# Print headers (text/html is the default)
print header(-type => 'text/html');
# Print <html>, <head>, <title>, <body> tags etc.
print start_html("Birthday");
# Your HTML body
print "Your birthday is $bday.
";
# Print </body></html>
print end_html();
```

Debugging Perl CGI Scripts

- Debugging CGI script is tricky - error messages don’t always come up on your browser
- Check if the script compiles
  ```bash
  $ perl -wc cgiScript
  ```
- Run script with test data
  ```bash
  $ perl -w cgiScript prod="MacBook" price="1800"
  ``
  ```text/html
  Content-Type: text/html
  <html>
  …
  </html>
  ```

How to get your script run

- This can vary by web server type
  ```bash
  http://www.cims.nyu.edu/systems/resources/webhosting/index.html
  ```
- Typically, you give your script a name that ends with .cgi and/or put it in a special directory (e.g. cgi-bin)
- Give the script execute permission
- Specify the location of that script in the URL

CGI Security Risks

- Sometimes CGI scripts run as owner of the scripts
- Never trust user input - sanity-check everything
- If a shell command contains user input, run without shell escapes
- Always encode sensitive information, e.g. passwords
  - Also use HTTPS
- Clean up - don’t leave sensitive data around
CGI Benefits

- Simple
- Language independent
- UNIX tools are good for this because
  - Work well with text
  - Integrate programs well
  - Easy to prototype
  - No compilation (CGI scripts)