Control

• So far, we've seen the basic building blocks of a Perl program:
  − The use of `Modern::Perl`
  − Output via the `print` or `say` functions
  − Interpolated vs. non-interpolated strings
  − Input via the `<>` operator; the use of `chop` and `chomp`
  − Variables / Scalars
  − Arithmetic operators (+, -, *, /, %, **, ++, --)
  − Other operators (...x, =)
• Now we need to start looking at how we control the execution of our code using programmatic control structures.

Simple Selection

• A selection structure allows a program to make a decision about whether or not to execute a particular piece of code.
• The core Perl language keeps its selection structures very simple by focusing on using only the `if` statement with a few variations.
### if statements

1. `#!/usr/bin/perl`
2. `use Modern::Perl;`
3. `print 'Would you like to see the 11:00 am showing of Casablanca? '
4.  `my $answer = <>;`
5.  `chomp $answer;`
6.  `if ($answer eq 'yes')`
7.  `{`
8.    `print "That'll be $8.50, please. " # ask him for money.`
9.  `}`

- Notice the use of `chomp` here to remove the newline from the `<>` before testing in line 6.
- Line 6 uses a conditional test to compare the scalar `$answer` to the string "yes" (non-interpolated, though "yes" would have tested the same thing).
- Line 6 uses the equality test operator `eq` to test for equality.
- Notice the use of curly braces in lines 6 and 9 for the `if` statement.
- Notice the use of `\$8` in line 8.

### if and alternative versions of if

The `if` Statement

```perl
if (CONDITION) {BODY}
```

executes BODY if the CONDITION is true

the curly braces { and } are required for the `if` statement

### Perl best practices: bracket placement

1. `#!/usr/bin/perl`
2. `use Modern::Perl;`
3. `print 'Would you like to see the 11:00 am showing of Casablanca? '
4.  `my $answer = <>;`
5.  `chomp $answer;`
6.  `if ($answer eq 'yes')`
7.  `{`
8.    `print "That'll be $8.50, please. " # ask him for money.`
9.  `}`

- Per the book "Perl Best Practices" (Google Books: [http://j.mp/perlbp](http://j.mp/perlbp)), page 9, Damian Conway insists that K&R (Kernighan & Ritchie) style bracketing and parenthesizing is the best way to code.
- K&R-style bracketing keeps the opening brace on the same line as the keyword (as in line 6 above) with the closing brace directly underneath the keyword itself.
- BSD/GNU-style bracketing keeps the braces on separate lines, as in the previous slide and most of the examples you’ll find throughout the class.
Backslashed characters

• In the last program examples, we used a backslash character in our output:

```
print "That'll be \$8.50, please.\n";
```

We backslashed the $ sign so that Perl wouldn't try and do variable interpolation on $8, thinking that $8 was a scalar (it is, but we don't talk about that this semester).

• What would the output have been if the \ had been left out?

• Could also have written:

```
print 'That will be $8.50, please.' . "\n";
```

or

```
print 'That will be $8.50, please.', "\n";
```

but not

```
print 'That will be $8.50, please.\n'; # why not?
```

Backslashed characters

• Perl `print()` Rule of Thumb #1:
  – For output, backslash weird characters (®, $, etc) to get their literal meaning (an ® symbol, a $ symbol, etc).

• Perl `print()` Rule of Thumb #2:
  – For output, backslash normal characters (n, r, t, etc) to get their weird meanings (linefeed, carriage return, tab, etc).

A little bit about expressions in Perl

• An expression is anything that, when evaluated, yields some value.

• The **CONDITION** of an if statement is an expression, such as:

```
if ( $answer eq 'yes' ) {}  
if ( 4 * 5 ) {}  
if ( $answer ) {}  
if ( chop $answer ) {}  
```

• What is the value of each expression above? For some of them, it depends on the value of $answer.

• Every condition or expression yields a particular value, which is usually a number or a string.
Expressions and truth

- Every expression, after being evaluated, is either true or false. The values below, and no others, are always false:

<table>
<thead>
<tr>
<th>Example</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 or 0.0</td>
<td>the number 0</td>
</tr>
<tr>
<td>&quot;&quot; or &quot;&quot;</td>
<td>an empty string</td>
</tr>
<tr>
<td>&quot;0&quot; or '0'</td>
<td>a string with just the 0 char</td>
</tr>
<tr>
<td>()</td>
<td>an empty list</td>
</tr>
<tr>
<td>undef</td>
<td>variables that don’t contain values</td>
</tr>
</tbody>
</table>

- All other values are true.

Examples of uninitialized variables

1. #!/usr/bin/perl
2. use Modern::Perl;
3. my $variable;
4. if ( !$variable ) { # no warning
5.  say "Variable has no value!";
6. }
7. say "Output: $variable!"; # warning
8. say "Math: ", $variable * 2; # warning
9. if ( !$variable ) {
10.  $variable = 42; # set a value!
11. }

- use Modern::Perl in the program catches some uses of uninitialized vars.
- Uninitialized variable errors are the most common type of error encountered in Perl programming and need to be handled if they crop up.

All inequality and equality operators

To test numeric values:

The expression ... evaluates to TRUE if the ...

<table>
<thead>
<tr>
<th>Expression</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x &gt; $y</td>
<td>number $x is greater than $y</td>
</tr>
<tr>
<td>$x &lt; $y</td>
<td>number $x is less than $y</td>
</tr>
<tr>
<td>$x &gt;= $y</td>
<td>number $x is greater/equal to $y</td>
</tr>
<tr>
<td>$x &lt;= $y</td>
<td>number $x is less/equal to $y</td>
</tr>
<tr>
<td>$x == $y</td>
<td>number $x is equal to $y</td>
</tr>
<tr>
<td>$x != $y</td>
<td>number $x is not equal to $y</td>
</tr>
</tbody>
</table>

To test string values:

The expression ... evaluates to TRUE if the ...

<table>
<thead>
<tr>
<th>Expression</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$word1 gt $word2</td>
<td>string $word1 comes after $word2</td>
</tr>
<tr>
<td>$word1 lt $word2</td>
<td>string $word1 comes before $word2</td>
</tr>
<tr>
<td>$word1 ge $word2</td>
<td>string $word1 after/equal to $word2</td>
</tr>
<tr>
<td>$word1 le $word2</td>
<td>string $word1 before/equal to $word2</td>
</tr>
<tr>
<td>$word1 eq $word2</td>
<td>string $word1 is equal to $word2</td>
</tr>
<tr>
<td>$word1 ne $word2</td>
<td>string $word1 is not equal to $word2</td>
</tr>
</tbody>
</table>
String ordering

- Is the word Hello less than or greater than hello?
- It’s actually less than. Hello comes before hello, because of the ASCII ordering. See an ASCII chart, where uppercase letters precede lowercase letters.
- At any Unix prompt:
  ```
  man ascii
  ```

When \texttt{lt} vs. \texttt{<} can go wrong

1. `#!/usr/bin/perl`
2. use Modern::Perl;
3. my $a = 62;  
4. my $b = 7;
5. if ( $a \texttt{lt} $b ) {  
6. print "Hey, $a is less than $b!\n";
7. }

- Operators in Perl force the data into specific constraints, so the string comparison \texttt{lt} will force $a and $b into string mode.
- What this means is that the integer value 62 will be treated as a string of characters (ASCII values 54 and 50) and integer value 7 will be treated as a string of characters (ASCII value 55).
- \texttt{lt} will then just compare 54 to 55 and come back with true.

When \texttt{eq} vs. \texttt{==} can go wrong

1. `#!/usr/bin/perl`
2. use Modern::Perl;
3. my $a = "one";
4. my $b = "two";
5. if ( $a == $b ) {  
6. print "Hey, $a and $b are equal!\n";
7. }

- Operators in Perl force the data into specific constraints, so the numeric == will force $a and $b into "numish" mode, and since there is no numeric value in either scalar (or string, or whatever), they both default to a value of 0, which results in the equation being true.
smart-matching with ~~

The ~~ Operator

Allows for correct equality matching between values. The actual cases which ~~ can be used in are wide and varied, but here are the pertinent ones for now:

<table>
<thead>
<tr>
<th>$a</th>
<th>$b</th>
<th>Implied match</th>
<th>Equiv Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Any</td>
<td>Num</td>
<td>numeric equality</td>
<td>$a == $b</td>
</tr>
<tr>
<td>Num</td>
<td>numish</td>
<td>numeric equality</td>
<td>$a == $b</td>
</tr>
<tr>
<td>Any</td>
<td>Any</td>
<td>string equality</td>
<td>$a eq $b</td>
</tr>
</tbody>
</table>

"numish" means a real number or a string that looks like a number

Using smart-matching ~~ to simplify eq vs. ==

1. #!/usr/bin/perl
2. use Modern::Perl;
3. my $a = "one";
4. my $b = "two";
5. if ( $a ~~ $b ) {
6.   print "Hey, $a and $b are equal!\n";
7. }

- No warnings and you get the equality test you were likely expecting.
- Unfortunately, there aren't any equivalent <= vs. le or other types of smart operators, so you just have to be careful making those types of comparisons.

if... else statements

1. #!/usr/bin/perl
2. use Modern::Perl;
3. print 'Would you like to see the 11:00 am showing of Casablanca? '; 
4. my $answer = < >;
5. chomp $answer;
6. if ( $answer ~~ 'yes' ) {
7.   print "That'll be $8.50, please \n";
8. }
9. else {
10.   print "Okay, but you're missing an excellent movie \n";
11. }

- If you don't type yes in lowercase letters here, the program assumes you don't want to see the movie.
- else clause gives you an alternative to doing only one thing.
• Now you have three options instead of just one or two.
• Use as many `elsif` statements as you need to achieve the desired result.