Programming in Perl

Week Two-

Data: types and variables

Control structures

Perl Data Types

- Question? What are computers good for?
  - Doing calculations
  - Storing and searching for information
  - Editing documents
  - Playing games
- It all comes down to playing with data!

Perl Data Types

- A “Scalar” is the most basic data type (called a “Primitive”) in Perl— every other type is made up of scalars
- Scalar can hold numbers, strings, and references
    - Perl automatically convert between numbers and strings
    - References “point” to other values

Scalars

- Scalar’s hold a single value
  - There are plural values, arrays and hashes, that we will look at later
- Perl has containers to hold scalars: scalar variables
- There are literal representations of numbers and strings

Numbers

- Perl can represent literal numbers in several different forms:

```
1234
-548
123_456_789
0377 (octal: 255 decimal)
0xFF (hex: 255 decimal)
3.1415926
1.66e-27
1.862E8 6.626E-34
```

Strings

- Strings are used to hold sequences of characters
- Strings can be any size: from zero sized until you fill up memory
  - A string can hold multiple lines– you can imbed a “newline” characters in the string.

Strings

- Two literal representations of strings in Perl:
  - Single Quoted— What you see is (almost) what you get

```
'Mark'
'is the best'
'Perl "teacher".'
'Don\'t forget to wash behind your ears!'
```

  - Double Quoted: also called interpolated string

```
"Hello, world!\n"
"you will say, \"I love Perl!\""
"This is an unprintable character -> \xFF"
```

- There is a third form of a literal string called a Bare Word that has several limitations
  - We will cover bare words later

Escape Characters

- Interpolated String understand “escape” commands
- Escape commands are a backslash (“\”) followed by the command

Character	Action	Character	Action
\n	Newline	\x7f	Hexadecimal version of Delete
\r	Return	\cD	Control-D
\t	Tab	\u	Force the next character to upper case
\b	Backspace	\l	Force the next character to lower case
\f	Form feed	\U	Force all characters to upper case until \E
\e	Escape	\L	Force all characters to lower case until \E
\a	Alarm (bell)	\E	End modification
\012	Octal version of Return	\Q	Quote regexp meta-character until \E

Escape Characters

- Case shifting interpolated strings:

```
"\uhello \LWORLD\E"        ➔      "Hello world"
```

- To put a backslash into a interpolated string, escape it!

```
"Use the Wall\\Schwartz transform!"
```

- If you put a backslash before a character that is not an escaped command, that character is inserted into the string

```
"This is an ’\A’"          ➔      "This is an ’A’"
```

Automatic Conversion of Strings to Numbers

- Perl automatically converts from number to strings, or string to numbers as needed
- A scalar is a number and string: you don’t need to worry if it is one or the other, just use it!

```
"15" + "22"          ➔        37
37 - "15"            ➔        22
"Mark is " . 33      ➔        "Mark is 33"
```

Automatic Conversion of Strings to Numbers

- A number is always converted to its decimal string representation
- The first number like thing in a string will be converted (leading white space is ignored)

```
"   15" + "22perl42"              ➔       37
```

Scalar Variables

- When you work with numbers, often you will right the number down on a slip of paper so that you can use it latter
- You do the same thing in programming languages by saving in Variables

Scalar Variables

- Perl uses a "meta-character" to mark its variables
- A scalar variable is a "$" followed by the variables name:

```
$[a-zA-Z_][a-zA-Z0-9_]*
```

- Examples:

```
$a
$_
$fred
$fred_and_barny
$A (not the same as $a)
```

Scalar Variables

- A scalar variable springs into existence the first time it is used
  - This is called “Autovivfication”
- If a variable is accessed before it has been assigned a value, the value of the variable is said to be undefined

Scalar Assignment Operators

- The value of a scalar variable is set with assignment operators

```
$<name> <assign op> <eval>
```

Operator	Action	Example
`=`	Plain old assignment. How boring.	`$answer = 42;`
`+=`	Add the value on the right to the value on the left.	`$bigger += 5;`
`-=`	Subtract the value on the right from the value on the left.	`$smaller -= 5;`
`*=`	Multiply the value on the right by the value on the left.	`$more *= 100;`
`/=`	Divide value on the right by the value on the left.	`$less /= 25;`
`%=`	Take the modulus on the right by the value on the left.	`$mod %= 3.14159;`
`**=`	Raise the value on the left to the power of the value on the right.	`$power **= 2;`
`.=`	Concatenate the string on the right to the value on the left.	`$addr .= "home";`
`x=`	Repeat the string value on the left by the number on the right.	`$again x= 5;`
`>>=`	Right bit shift the value on the left by the value on the right.	`$right >>= 2;`
`<<=`	Left bit shift the value on the left by the value on the right.	`$left <<=2;`
`&=`	Bit AND the value on the left by the value on the right.	`$and &= 0xFF;`
`|=`	Bit OR the value on the left by the value on the right.	`$or /= 0xAA;`
`^=`	Bit NEGATE the value on the left by the value on the right.	`$xor ^= 0x55;`

Short Digression on Style

- *How* you write your code is as important as *what* you write
  - You may write the most efficient, amazing, code in the world, but if you can't read, you can't maintain it
- As we work our way through Perl, we will be discussing Programming Style
  - Mostly, it will be general, and will apply to any programming language, but we will get specific and talk Perl Style as will
  - You can check the Perl Style manual on PerlDoc.com web site for more details
- I will be using "*The Elements of Programming Style*" by Kernighan and Plauger to augment the style discussions

Naming Conventions

- "*Choose variable names that won't be confused*" - EofPS
- Try to pick names that are descriptive of what they are used for- Use nouns

```
$name = 'Mark';
$n = 'Joe';
$line = <FILE>;
$input = <FILE>;
```

- Keep the name short

```
$a_long_variable_name_like_this_can_be_hard_to_read
```

- Be consistent

```
$LastName
$last_name
```

Expressions

- Once you have some data, and you have it stored somewhere, you need to be able to do something with it
- Expressions are the something: they are evaluated to return a value
- Expressions are made up of literal values, variables, operators, and/or functions

```
$foo = 3 + 5;
$bar = ($foo + 7)/5;
```

Operators

- Numeric Binary Operators

```
<eval> <op> <eval>
```

Operator	Description	Example
`+ -`	Numeric additive operators. Adds or subtracts two numbers. Strings are converted to numbers as needed.	`$bigger = $a + $b;`
`* / %`	Numeric multiplicative operators. Multiply, divide and take the modulus of two numbers. Strings are converted to numbers before the operation. The % operator converts the numerator and denominator to integers before finding the remainder.	`$less = $a/$b;` `$remains = $a % $b;`
`**`	Exponentiation Operator. Usage: num**exp.	
`,`	The comma operator. In a scalar context it evaluates its left argument, throws that value away, then evaluates its right argument and returns that value. In a LIST context (don't ask, it'll come later!), it's just the list argument separator, and inserts both its arguments into the list.	`$c = $a++, $b;` `# $a is incremented` `# $c = $b`

Operators

- String Binary Operators

Operator	Description

`.`	String concatenation operator. Usage: $string . $string. This returns a concatenated string. Numbers are converted to strings before the concatenation.
	
	`$city='Phoenix'; $state='AZ'; $zip=85044;`
	`$address = $city . ', ' . $state . ' ' . $zip;`

`x`	String/list repetition operator. When the left-hand side is a string, the operator returns a string that is repeated by the number on the right-hand side. If the left-hand side is a LIST enclosed in parentheses, the operator returns an array that is the number on the right-hand side in length and each element is equal to the LIST.
	
	`print 'The answer is' . '-'x30 . '42!';`
	`The answer is------------------------------42!`

Variable interpolation of scalars into strings

- Works on double-quoted strings— sometimes called “double-quote interpolation” or “interpolated strings”
- Any legal scalar variable name is replaced by its current scalar value inside of a double-quoted string

```
$c = "Hello";
$b = "$c world";            # $b is now "Hello world"
```

- Undefined values are replaced with the empty string

```
$c = "$nowhere foo";        # " foo" if $nowhere is undefined
```

Variable interpolation of scalars into strings

- Interpolation happens as the scripts runs (“at run time”), so the same text could be expanded using the current value of the variable when the statement is executed

```
$a = "hello";
$b = "What I meant to say was '$a'"      # What I meant to say
                                         # was ’hello’
$a = "good-bye!";
$b = "What I meant to say was '$a'";     # What I meant to say
                                         # was ’good-by’
```

Variable interpolation of scalars into strings

- To prevent variable interpolation, preceded the “$” with a backslash:

```
$c = "The variable name is \$a";         # no expansion
```

- If you really don’t want any backslash or variable interpolation, use single quotes instead of double quotes. Single quote strings are faster and easier to read

```
"The variable \$total holds the dollar value of the ca\$h"
’The variable $total hold the dollar value of the ca$h’
```

Generating Output with `print`

- print takes a single scalar argument, sends it (as a string) to standard output (typically the user’s terminal)

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

- Can also be written with parentheses

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

- Can also take a bunch of scalars separated by commas

```
$world = "world";
print "hello", ’ ’, $world, "\n";
```

Getting input with `<STDIN>`

- Special operator `<STDIN>` is replaced with the next line from standard input
- Includes all characters entered on the user's terminal, including the trailing newline character
- Can be used with output operators to provide interaction with user

```
print "What is your name? ";
$name = <STDIN>; # reads a line from the terminal
print "Hello $name Are we not having fun?\n";
```

Using `chomp()` to remove newline

- You can remove the newline simply by using the `chomp()` operator

```
print "What is your name? ";
$name = <STDIN>; # reads a line from the terminal
chomp $name;
print "Hello $name Are we not having fun?\n";
```

- `chomp()` removes the trailing newline, if there is one

Arrays

- Arrays are a sequentially ordered set of scalars
  - Each scalar is called an element of the array
- The index into an array is an integer, starting with `0`

42	3.1415	"Mark"	undef	"end\n"
0	1	2	3	4

- Arrays have no predefined size. Like strings, they can grow to the limit of memory

Array Literals

- An array literal is a left parentheses, "(", with zero or more comma separated scalars, ending with a right parentheses, ")"

```
(42, 3.1415926, "Mark", undef, "end\n")
() # The empty array
($name, $street, $city, $zip)
```

- An array literal is called a LIST
- Many functions take a LIST as its argument
  - If the function expects a LIST as its argument, the "()" are not necessary

```
print $a, $name, “\n”;
print($a, $name, “\n”);
```

Array Assignment

- Unlike scalar variables, there is only one array assignment operator— “=”

```
@loc = (1.2, 3.5);
```

- The scalar value returned from an array assignment is the number of array elements assigned

```
$num = (@new_loc = @loc); # $num has the value 2
```

- The value returned from an array is usually a LIST containing all of the array elements

```
($x, $y) = @loc;
```

Array Element Access

- Single elements of an array variable are scalars, and are accessed like a scalar
  - When you access a single element from an array, you must use the “$” prefix!
- You access an element by putting its index inside of “[]”
  - This syntax is called an array subscript

```
@a = ("first","second","third");
$b = $a[0]; # $b gets "first"
$c = 2;
$d = $a[$c]; # $d gets "third”
$a[1] = "two"; # changes "second" to "two"
```

The `push()` and `pop()` operators

- The `push()` and `pop()` operators do things to the end of an array (the end with the larger index)
- `pop()` removes the end of the array, returning the removed element

```
@a = (1,2,3,4,5);
$b = pop(@a); # $b is 5, and @a is now (1,2,3,4)
$c = pop(@a); # $c is 4, and @a is now (1,2,3)
@empty = ();
$d = pop(@empty); # $d gets undef, @empty unchanged
```

The `push()` and `pop()` operators

- `push()` adds a new element (or elements) onto the end of the array (even an empty array)

```
@x = (); # initialize to empty array
push(@x,1); # @x now (1)
push(@x,2); # @x now (1,2)
push(@x,3,4,5); # @x now (1,2,3,4,5)
@y = (6,7,8); push(@x,@y); # @x now (1,2,3,4,5,6,7,8)
```

The `shift()` and `unshift()` operators

- `shift()` and `unshift()` are to the start of an array what `pop()` and `push()` are to the end

```
@x = (6,7,8);
unshift(@x,5); # @x now (5,6,7,8)
unshift(@x,1,2,3,4); # @x now (1,2,3,4,5,6,7,8)
$y = shift(@x); # $y gets 1, @x now (2,3,4,5,6,7,8)
```

- `shift()` and `unshift()` are fairly efficient—they don’t copy the array, but merely shuffle some indices— however, if you can choose between `push`/`pop` and `shift`/`unshift`, choose push/pop for efficiency
- `shift()` is often used in stepping through an array

Array Slices

- A list of scalar elements from a single array is called a slice, and have a special representation

```
@a = ($fred[3],$fred[4],$fred[10])# three elements from @fred
@a = @fred[3,4,10]                # the same thing as a slice
```

- Because you are trying to get a LIST from the array, you must prefix the array variable with "@"
- You can assign to slices or you can return the values:

```
@fred[3,4,10] = (6,7,8);          # give them three values
print "now it is @fred[3,4,10]";  # prints "now it is 6 7 8"
```

- The subscript is really a list expression:

```
@subs = (3,4,10);
print "now it is @fred[@subs]";   # same as above
```

Array Slices

- A single element in the subscript gives a slice of one element, not a scalar!

```
@a[3] = @b; # same as ($a[3]) = @b, not $a[3] = @b;
```

- A list of literals can be used in place of the array (sometimes called a literal slice):

```
("mon","tue","wed","thu")[1,3] # same as ("tue","thu")
```

- The parentheses in literal slice are required

Hashes

- A hash is like an array, but with indices (keys) of arbitrary strings

42	3.1415	"Mark"	undef	"end\n"
answer	PI	me	far	end

- Hash elements have no particular order in the hash—merely a collection of key-value pairs
- Keys and values are both scalars (but keys are always converted to strings)
- Hashs can have any number of elements, from zero to all of memory, just like arrays and strings

Hash Variables

- You can refer to a complete hash variable with the percent (“%”) prefix

```
%a
%b
%a_very_long_variable_name
```

- Once again, the names come from a separate name space (`%a` has nothing to do with `$a`, or `@a`)
- A hash is rarely used as a whole
- Access to hashs is by using a subscript enclosed in "`{}`"

```
$some_hash{"foo"} = 35;
$bar = "foo";
print $some_hash{$bar}; # prints 35
```

Hash Variables

- Like normal arrays, elements spring into existence by assignment
- Note that `$some_name{$some_key}` has nothing to do with `$some_name`, `$some_name[$index]`, or `$some_name`
  - Scalars, arrays, and hashs can have the same name because they all have different name spaces

Literal Representation of a Hash

- A hash can be initialized with a LIST, just like an array

```
%some_hash = (
   'answer  '        , 42,
   'PI'              , 3.1425926,
   'me'              , 'Mark',
   'far'             , 1.72e30,
   'end'             , "end\n"
);
```

- A hash can be "unwound" into a LIST for assignment to an array

```
@some_var = %some_hash; # unwinds the hash into @some_var
```

- The key/value pairs are returned in an arbitrary order

Literal Representation of a Hash

- The => operator can be used to make it even easier to create literal hashes

```
%some_hash = (
  answer          => 42,
  PI              => 3.1425926,
  me              => "Mark",
  far             => 1.72e30,
  'end'           => "end\n"
);
```

- “=>” is like the “,” except that it treats the characters, delimited by whitespace, to its right as a string. This kind of string is called a "Bare Word"

Hash Operators

- The `keys(%some_hash)` operator gives an array of the keys while the `values(%some_hash)` operator gives the corresponding values

```
%list = ("a",1,"b",2,"c",3);
@k = keys %list;   # @k is some permutation of (“a”,”b”,”c”)
@v = values %list; # and @v is the corresponding
                   # values from %list (1,2,3)
```

- Don’t change the hash between using `keys()` and `values()` or you will get unpredictable results!

Hash Slice

- Hash slices are similar to an array slice

```
# three elements from %score
($score{"fred"},$score{"barney"},$score{"deno"})
# equivalent form
@score{"fred","barney","deno"}
```

- Hash slices can be used as variables or to return values:

```
@score{"fred","barney","deno"} = (205,195,30);
```

- The subscript is a really a list expression:

```
@subs = ("fred","barney","deno");
@score{@subs} = (205,195,30); # same as above
```

Scalar and Array Context

- Parts of an expression are expected to be a scalar or an array

```
$x = ...; # scalar assignment, so right side must be a scalar
@y = ...; # array assignment, so right side must be an array
```

- Type of a value is called the context
- Some operators return different values in different contexts, like array assignment

```
@b = @a; # @a copies the elements from array a to array b
$b = @a; # $b gets the number of elements
$c = (@a = @b); # $c gets the number of elements copied
@d = (@a = @b); # @d gets the actual elements
```

Scalar and Array Context

- All operators have a specific context (scalar or array) for their operands— keep this in mind
- Some array operators have no scalar value (like `sort()`)
- A scalar value used in an array context is automatically promoted to an array value of one element

```
@a = 2+3; # @a = (5)
```

Scalar and Array Context

- In the same way, `undef` used in an array context results in a single-element list of `undef`

```
@b = undef; # same as @b = (undef)
```

- There is no automatic conversion of LIST to scalar— the value is determined using the array to scalar rules

```
$b = ("foo", "bar", "baz");      # $b gets 3
```

- You can force a scalar context by using the `scalar()` operator

```
print "There are " . scalar(@a) . "elements in the array";
```

References

- A scalar can hold a number or a string
  - They can also hold references
  - References are a way to use a variable indirectly
  - A reference “points” to the value of variable without knowing the name of the variable
  - A reference is always contained in a scalar variable (or a scalar element in an array or hash)
- We will learn more about references in Week Four

Control Structures

- Statements are executed in sequence from the beginning to the end of a program
  - This is know as flow of control of the program
  - This is adequate only for extremely simple programs

```
$name = 'Mark';
print "$name\n";
```

- To create useful programs, you need to be able to execute different sections of your program depending on the conditions in the program at a particular time, as the program is running
- This is handled with control structures and statement blocks

Statement Blocks

- Perl's statement blocks are an open curly brace ("}"), one or more statements, then close curly brace ("}")

```
{
    some statement;
    some other statement;
    another statement;
}
```

- Typically, a block is part of a larger control structure but they can be used in a script by themselves
  - Statement blocks that appear by themselves are called Naked Blocks
  - We will see how this is useful later
- A semicolon on the final statement in a block is optional (but put it there anyway!)

Selection Statements

- Often, you will want to modify the flow of control in a program, selecting different parts of a program depending on a condition
- Perl provides this with an `if` statement

```
if (condition) {
  statement;
  statement;
}
```

- The block is executed if the condition expression is true, otherwise it is skipped

Selection Statements - `if`

- Braces are always required
- Control expression is computed as a string value (numbers are converted to strings)
  - False is the empty string, or the string consisting of the single character “`0`” (zero)— anything else is true
- Any expression can be used as the condition to be tested

```
$foo = "The trilly-tromp did slyther through the slomp.\n"
if ($foo) {
    print $foo;
}
```

Relational Operators

- Perl provides several operators that test one value against another

Operator		**Description**
String	**Numeric**	
`$a cmp $b`	`$a <=> $b`	Comparisons. These operators return `-1` for less-than, `0` for equal, and `1` for greater than. `cmp` is used for strings and `<=>` for numeric
`$a eq $b`	`$a == $b`	Equal to comparison. `eq` for strings and `==` for numeric
`$a ne $b`	`$a != $b`	Not equal to. `ne` for strings and `!=` for numeric
`$a gt $b`	`$a > $b`	Greater than. `gt` for strings and `>` for numeric
`$a lt $b`	`$a < $b`	Less than. `lt` for strings and `<` for numeric
`$a ge $b`	`$a >= $b`	Greater than or equal to. `ge` for strings `>=` for numeric
`$a le $b`	`$a <= $b`	Less than or equal to. `le` for strings `<=` for numeric

Logical Operators

- The Logical Operators are "Short Circuit"
  - They only evaluate as much as they need to generate the result
  - The return value is always the last thing evaluated

Operator	**Description**	Example
`&& and`	Logical And Operators: if the first operand is false, the operation returns false. If the first operand is true, the second operand is returned	`$a && $b`
`|| or`	Logical Or Operators: if the first operand is true, the operation returns the first operand. If the first operand is false, the second operand is returned	`$a || $b`
`! not`	Logical Negation. Inverts the truth of expression. True becomes false and false becomes true	`!$a`

Control Statements - `else`

- The `if` statement allows you make choices of blocks by using the `else` clause

```
if (condition) {
    true statement;
} else {
    false statement;
}
```

- If the condition is true, do the true block, otherwise do the false block

Control Statements - `elsif`

- Multi-way selections can be done using the `elsif` clause
  - That is `elsif` with no 'e'

```
if (condition_1) {
    condition_1_true;
    condition_1_true;
} elsif (condition_2) {
    condition_2_true;
    condition_2_true;
} elsif (condition_3) {
    condition_3_true;
    condition_3_true;
} else {
  nothing_true;
}
```

Control Statements - `elsif`

- By using simular test conditions, you can make a case statement

```
if ($a == 1) {
    print "a is 1\n";
} elsif ($a == 3) {
    print "a is 3\n";
} elsif ($a == 4) {
    print "a is 4\n";
} else {
    print "something else!\n";
}
```

Control Statements - `unless`

- The unless statement is like the if statement, except that it reverses the condition

```
unless (condition) {
  condition_false;
  condition_false;
}
```

- You use this to increases readability
- Also, TIMTOWTDI

Using Logical Operators as Control Statements

- Because the logical operators work by short circuit, you can use them as control statements!

```
open FILE, $file or die "Could not open file!";
```

- Read this as "open the file or die!"

```
unless (open FILE, $file) {
    die "Could not open file!";
}
```

- This is a Perl idiom, you will see it in most programs
  - TIMTOWTDI again
- "*Use the 'telephone test' for readability*" - EofPS
  - If someone can understand you code when read aloud over the telephone, then it's clear enough

Looping Control Structures - `while`

- Many times, you want to repeat a block of code. Perl has several Looping Control Structures that do just that

```
while (condition) {
    body;
    body;
    body;
}
```

- The condition is evaluated before the first iteration—possibly skipping the body entirely.
- Like `if`, the block curly braces are required

Looping Control Structures - `while`

- The while loop my also have a optional `continue` block

```
while (condition) {
    loop;
} continue {
    block;
}
```

- The `continue` block is executed each time the while loop continues to be executed
  - It is executed after the loop has been executed

Looping Control Structures - `until`

- Replacing the `while` with `until` reverses the sense of the condition

```
until (condition) {
    loop_if_flase;
}
```

Looping Control Structures - `for`

- Perl `for` is like C's `for` statement:

```
for (initial; condition; increment) {
  body;
  body;
}
```

- ...which is more or less equivalent to…

```
initial;
while (condition) {
  body;
  body;
} continue {
  increment;
}
```

Looping Control Structures - `for`

- The `for` statement is typically used to step through computed iterations

```
for ($i = 1; $i <= 10; $i++) {
  print “I can count to $i!\n”;
}
```

- The `initial`, `condition`, or `increment`, can be empty
- An empty `condition` means loop forever

```
for (;;) {
  body;
}
```

- The initial, test, and increment parts must be single (but possibly complex) expressions
  - You can’t use blocks, but you can use the “`,`” operator to stack up several expressions

Looping Control Structures - `foreach`

- The `foreach` statement steps through an array (or LIST value), assigning a variable to each value in the array

```
foreach $control_variable (@some_array) {
  body;
  body;
}
```

- The control variable is local to each foreach statement
  - The original value of the variable is restored after the loop has finished
- If the variable is omitted, the system variable $_ is used.

```
foreach (@thing) {
  # using $_
}
```

- The words `for` and `foreach` are interchangeable (Perl figures it out)

Looping Control Structures - `foreach`

- Changing the value of the control variable also changes the corresponding element of the array

```
@a = (1,2,3,4,5);
foreach $i (@a) {
  $i = $i + 3;
}
# @a is now (4,5,6,7,8)
```

- The current index is not available, but you can compute it easily

```
@a = ("hello","world","this is","a test");
$index = 0;
foreach $s (@a) {
  print "element $index is $s\n";
  $index++;
}
```

Loop Control Statements

- Perl provides several ways to control loops
  - `last` breaks you out of the loop
  - `next` end the current loop and starts the next
    - If you have a continue block, it is executed
  - `redo` ends the current loop and repeats it
    - The continue block is skipped

```
while (1) {
    $line = <STDIN>;
    if ($line eq 'q') {
        print "Goodbye!\n";
        last;
    }
}
```

Statement Modifiers

- An statement can be followed with a modifier to provide more compact notation

```
$a = $b if $a < $b;                     # like:
                                        # if ($a < $b) { $a = $b; }
print "bigger!\n" unless $num <= 4;     # like:
                                        # unless ($num <= 4) {
                                        #     print "bigger!\n";
                                        # }
print ++$n, " " while $n < 10;          # like:
                                        # while($n<10){
                                        #     print ++$n,” “;
                                        # }
$i *= 2 until $i > $j;                  # like:
                                        # until ($i > $j) {
                                        #     $i *= 2;
                                        # }
```

Statement Modifiers

- Only a single expression allowed on the left and right—no cascading
- Reverses the order of left to right evaluation—the modifier expression is always evaluated first

More Programming Style

- "*Write clearly - don't be too clever*" - EofPS
  - Don't do something just because it's cool
- "*Say what you mean, simply and directly*" - EofPS
  - Don't try to do to many things at one time
- Pick a consistent coding style and stick with it
  - Indent blocks 4 spaces
  - "cuddle up" braces
- "*Avoid the Fortran arithmetic IF*" - EofPS

Home Work

- Read Chapters 4, 6, 11
    1. Write a program that asks for a weight in pounds and displays the equivalent weight in kilograms (1 kilogram equals 2.2 pounds).
    2. Write a program that calculates the gross pay of an employee. The program should ask for the hourly rate of the employee, how many regular hours, and how many overtime hours the employee worked. Pay for overtime hours should be calculated at time and a half (1.5 times the regular hourly rate).
    3. Write a program that prompts for and accepts up to twenty numbers (i.e., provide a way for a user to enter less that twenty values) and prints out these values in reverse order.
    4. Modify the above program to remove any duplicate values from the list, while still telling how may times a given value was entered.