Control Structures

if-elsif-else

- semantically the same as C/C++
- syntactically, slightly different.

```
if ($a > 0){
  print "\$a is positive\n";
} elsif ($a == 0){
  print "\$a equals 0\n";
} else {
  print "\$a is negative\n";
}
```

- brackets are *required*!

unless

- another way of writing **`if (!…) {…}`**
- analogous to English meaning of "unless"
- **`unless (EXPR) BLOCK`**
  - "do BLOCK unless EXPR is true"
  - "do BLOCK if EXPR is false"
- can use **`elsif`** and **`else`** with unless as well

while/until loops

- **`while (EXPR) BLOCK`**
  - "While EXPR is true, do BLOCK"
- **`until (EXPR) BLOCK`**
  - "Until EXPR is true, do BLOCK"
  - "While EXPR is false, do BLOCK"
  - another way of saying **`while (!…) {…}`**
- again, brackets are *required*

while magic

- If and only if the only thing within the condition of a while loop is the readline operator:
  `while (<$fh>) { }`
- perl automatically translates this to:
  `while (defined($_ = <$fh>)) { }`
  - **`$_`** holds the line that was just read.
  - When **`<$fh>`** returns **`undef`** (ie, file completely read), loop terminates.
- This does NOT happen anywhere else!
  `<$fh>;`
  - Reads a line and throws it away, does NOT assign to $_

do

- Execute all statements in following block, and return value of last statement executed
- When modified by while or until, run through block once before checking condition

```
do {
  $i++;
} while ($i < 10);
```

for loops

- Perl has 2 styles of **`for`**.
- First kind is virtually identical to C/C++
- for (INIT; TEST; INCRMENT) { }
- ```
  for (my $i = 0; $i < 10; $i++){
    print "\$i = $i\n";
  }
  ```
- yes, the brackets are required.

foreach loops

- Second kind of for loop in Perl
  - no equivalent in core C/C++ language
- **`foreach VAR (LIST) {}`**
- each member of LIST is assigned to VAR, and the loop body executed
- ```
  my $sum;
  foreach my $value (@nums){
    $sum += $value;
  }
  ```

More About for/foreach

- **`for`** and **`foreach`** are synonyms
  - Anywhere you see "for" you can replace it with "foreach" and viceversa
    - Without changing ANYTHING ELSE
  - they can be used interchangeably.
  - usually easier to read if conventions followed:
    - **`for (my $i = 0; $i<10; $i++) {}`**
    - **`foreach my $elem (@array) {}`**
  - but this is just as syntactically valid:
    - **`foreach (my $i = 0; $i<10; $i++) {}`**
    - **`for my $elem (@array) {}`**

foreach triviata

- **`foreach VAR (LIST) {}`**
- while iterating through list, VAR becomes an *alias* to each member of LIST
  - Changes to VAR within the loop affect LIST
- if VAR omitted, $_ used instead
  - it too is an alias
-
```
@array = (1, 2, 3, 4, 5);
foreach (@array) {
  $_ *= 2;
}
```
- @array now L (2, 4, 6, 8, 10)

foreach loop caveat

-
```
my $num = 'alpha';
for $num (0..5) {
  print "num: $num\n";
}
```
- Upon conclusion of the loop, $num goes back to 'alpha'!
  - The for loop creates its own lexical variable, even though you didn't specify `for my $num (0..5) { … }`
- For this reason, it is always preferred to *explicitly* declare the variable lexical to the loop, to avoid the possible confusion.

Best Practice

- There is *rarely* any need to use C-style for loops in Perl
- If you want to iterate over a count value:
- **`foreach my $count (0..$total) { }`**
- **`foreach my $i (0..$#array) { }`**
- Only time you need a C-style for loop is to increment by something other than 1:
- **`for (my $v=0; $v < $tot; $i+=3) { }`**

`foreach (<$fh>){ }` VS `while (<$fh>){ }`

- Both constructs appear to do the same thing.
  - Assign each line of **`$fh`** to **`$_`**, execute loop body
- The difference is internal
  - **`foreach`** takes a LIST
    - evaluates **`<$fh>`** in list context, once
  - **`while`**'s condition is **`defined($_ = <$fh>)`**
    - evaluates **`<$fh>`** in scalar context, repeatedly
- **`foreach`** will read the entire file into memory at once
  - each element of resulting list is then assigned to **`$_`**
- **`while`** will read the file line by line, discarding each line after it's read
  - FAR more efficient. Always use **`while (<$fh>) { }`**

Reading it in English

- Perl has a cute little feature that makes simple loop constructs more readable
- If your **`if`**, **`unless`**, **`while`**, **`until`**, or **`foreach`** block contains only a single statement, you can put the condition at the end of the statement:
- **`if ($a > 10) {print "\$a is $a\n";}`**
- **`print "\$a is $a\n" if $a > 10;`**
- Using this modifier method, brackets and parentheses are unneeded
- This is syntactic sugar – whichever looks and feels right to you is the way to go.

Loop Control – next, last, redo

- **`last`** – exit innermost loop
  - equivalent of C++ **`break`**
- **`next`** – begin next iteration of innermost loop
  - (mostly) equivalent of C++ **`continue`**
- **`redo`** – restart the current loop, without evaluating conditional
  - no real equivalent in C++
- Note that Perl does not consider **`do`** to be a looping block. Hence, you cannot use these keywords in a do block (even if it's modified by while)

Breaking Out of More Loops

- **next**, **last**, **redo** operate on innermost loop
- Labels are needed to break out of nesting loops

```
TOP: while ($i < 10){
  MIDDLE: while ($j > 20) {
    BOTTOM: for (@array){
              if ($_ == $i * $j){
                last TOP;
              }
              if ($i * 3 > $_){
                next MIDDLE;
              }
            }
            $j--;
          }
          $i++;
        }
```

goto

- yes, it exists.
- No, don't use it.