Controlling Flow with Perl

- Perl has the usual `if`, `while`, and `for` control-flow mechanisms that exist in C/C++/Java

- Unlike other languages, Perl *requires* the use of braces (i.e., `{` and `}`), so:

```
if ( $count > 20 )
{
    die 'Count has exceeded its limit!';
}
```

- is legal Perl, whereas:

```
if ( $count > 20 ) die 'Count has exceeded its limit!';
```

- is not legal, resulting in the following error message from the Perl interpreter:

```
syntax error at test.pl line 1, near ") die"
Execution of test.pl aborted due to compilation errors.
```

`if` Syntax

- We have already seen the simplest form of `if` (on the last slide)
- As you would expect, we can also have an `else` part:

```
if ( length( $line ) >= 80 )
{
    print "Line length is wider than the standard text-mode screen.\n";
}
else
{
    print "Line length is okay - it'll fit.\n";
}
```

- The keyword `elsif` is used whenever we have a mutually exclusive series of tests, and can be used to simulate a switch statement, which is nice to know, because Perl *does not have* a switch statement!

`if`, `elsif`, `else` Example

```
if ( $a == 1 )
{
    print "The value of scalar a is: $a.\n";
}
elsif ( $a == 2 )
{
    print "The value of scalar a is: $a.\n";
}
elsif ( $a == 3 )
{
    print "The value of scalar a is: $a.\n";
}
else
{
    print "The value of scalar a is something else.\n";
}
```

- Note that the keyword `unless` can be used anywhere an `if` appears, and has the effect of *negating the condition* being tested

Doing `if` on One Line

- We have already seen what follows in C/C++/Java, and it exists in Perl also
- The following `if` statement:

```
if ( length( $a ) >= length( $b ) )
{
  $longer = $a;
}
else
{
  $longer = $b;
}
```

- can be written more compactly as:

```
$longer = ( length( $a ) >= length( $b ) ) ? $a : $b;
```

Looping in Perl

- Our very first Perl script introduced the `while` statement, which keeps executing something while the condition being tested is *true*
- Perl allows you to use `until` anywhere you use `while`, and the effect is to keep executing something while the condition being tested is *false* (i.e., until it is true)
- Like the `if`, the curly braces are *required* with loops
- You can, again, just like with `if`, use `while` and `until` as qualifiers to a single statement in Perl:

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

print while (<>);
```

- is a shorter, *but equally valid*, version of our first Perl script (now it's only 2 lines long!).

Extra Looping Controls

- Perl provides some specific statements which can be used to fine-tune the behaviour of loops:
  - `last`: causes an immediate exit from the current loop
  - `next`: causes the current iteration of the loop to be abandoned, with control jumping back to the controlling `while` or `until` statement (at the top of the loop)
  - `redo`: causes the current iteration of the loop to be abandoned, jumps to the start of the loop, and starts re-executing the code without testing the loop condition

More Looping Controls

- `continue`: identifies a block of code at the end of the loop that is executed at the end of each loop *before the next iteration*
- However, if a `redo` or `last` command is executed within the loop, Perl will *skip* the `continue` block of code
- The `next` command will *always* execute the `continue` code before returning to the top of the loop to test for the next iteration
- Sometimes it is useful to purposefully create an infinite loop, and it's easy to do so in Perl:

```
while ()
{
    # Do something forever ...
}
```

A Looping Example

- This example is taken from Nigel Chapman's textbook
- We will construct a simple *command interpreter* which controls the value of a single variable (not very useful, but it will illustrate what we've seen so far)
- The user of this program can issue the following commands:
  - `up` : adds one to the value
  - `down` : subtracts one from the value
  - `zero` : resets the value to 0
  - `quit` : exits the program
  - `!` : repeats the last command (if it was legal)

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

# Set some variables to their initial state.  We are just being nice, as
# we don't have to do this in Perl.

$n = 0;
$last_cmd = '';

# Tell Perl that output can be AUTOFLUSHED, i.e., no need to wait for a
# new-line prior to writing output.  Note: if we had included 'use English';
# at the top of our script, we could refer to this special variable in
# its English form: $OUTPUT_AUTOFLUSH.

$| = 1;

# Look for some initial input from the user.  Note that the output does not
# include a new-line at the end, which is what we want.

print "\nThe value is: $n\nEnter a command?  ";
chomp( $_ = <STDIN> );    # Note: $_ is $ARG if we 'use English;'.

# Enter an infinite loop, and process commands until done.
while ()
{
     # We process the easy commands first.

     if ( $_ eq 'up' )
     {
          ++$n;
     }
     elsif ( $_ eq 'down' )
     {
          --$n;
     }
     elsif ( $_ eq 'zero' )
     {
          $n = 0;
     }
     elsif ( $_ eq '!' )
     {
          # If we have a last command, we reprocess it, otherwise we have
          # no previous command.  Note: if a string is empty, it is false.
```

```
            if ( $last_cmd )
            {
                $_ = $last_cmd;
                redo;
            }
            else
            {
                print "No previous command to redo, sorry.\n";
                next;
            }
        }
        elsif ( $_ eq 'quit' )
        {
            # We are done, so we use 'last' to exit from the loop.

            last;
        }
        else
        {
            # Tell the user that we do not know the command entered.

            print "Unknown command << $_ >>.\n";
            print "Use either up, down, zero, !, or quit.\n";
            next;
        }

        # Remember the last command (only if it was valid).

        $last_cmd = $_;
    }
    continue
    {
        # We always ask the user for another command before returning to the
        # top of the loop, which is why this code is in the 'continue' block.

        print "\nThe value is: $n\nEnter a command?  ";

        chomp( $_ = <STDIN> );
    }
```

Looping a Number of Times

- Perl supports the `for` looping construct, and it behaves exactly as it does in C/C++/Java:

```
for ( $i = 1; $i < 10; ++$i )
{
    print 'The value of $i is: ', $i, "\n";
}
```

- Perl extends the notion of a `for` loop to provide a `foreach` statement

- The `foreach` statement is used with *arrays* and *lists*, and we'll see how to use it later on in this course

Block Expressions

- Perl allows you to write a sequence of statements (which produce an expression result) anywhere that an expression is expected
- This is accomplished with the `do` statement (which is *not* a loop)
- Consider the following:

```
while ( do
    {
        print "\nThe value is: $n\nEnter a command?  ";
        chomp( $_ = <STDIN> );
        $_ ne 'quit';
    } )
```

- If we had used this code with our simple interpreter, we would remove the associated `last` statement and the `continue` block (as they are no longer needed)

Introducing Subroutines

- Call them what you like (subroutines, routines, functions, procedures, methods ...), Perl has such a mechanism

- Here's a simple (stupid) example which shows the basic structure:

```
sub stupid {
   print "Hi!  I'm stupid.\n";
}
```

- So, subroutines in Perl are introduced by the keyword `sub`, followed by a subroutine name, followed by a block of code to execute

- Perl is pretty easy going about subroutines - they can appear anywhere within your script file, and do not need to be declared before your code calls them

More Subroutine Stuff

- Having seen subroutines in other languages, we know to expect more from them
- We want to be able to *return results*, use *variables* that are *local* in scope to the subroutine, and, we want to be able to *pass arguments into* the subroutine
- Not to be outdone, Perl lets you do all these things
- We will look at arguments after we have seen arrays, as that's the mechanism Perl relies on to pass arguments into subroutines
- Getting results and using local variables is real easy

Getting Results from Subroutines

- Results can be generated in one of two ways
- We can use an explicit `return` statement:

```
return( 42 );
```

- which can appear anywhere in the subroutine
- We can also rely on Perl's default behaviour, that is, if a return statement is not provided, Perl will treat the last statement of the subroutine as an expression, and *return the result of the evaluated expression* as the subroutines result:

```
sub stupid_too {
  $phrase = q[Hi!  I'm stupid, too.];
}
print stupid_too, "\n";
```

Local Variables within Subroutines

- We use Perl's built-in keyword `my` to indicate that a variable used within a subroutine is local to the subroutine
- All other variables are *global*
- Watch out for variables declared as `local` within Perl subroutines - this is a carry-over from earlier versions of Perl, and has the effect of making the variable declared as `local` to be available in its own subroutine, as well as in any subroutines called from within its own subroutine (which isn't really local, is it?)
- Always use `my` over `local`, but remember that a lot of older, existing Perl scripts make extensive use of `local` (because `my` wasn't available prior to Perl version 5)

Example of Locals and Globals

- The following script:

```
sub really_stupid {
    $phrase_g       = q[Hi!  I'm really_stupid's phrase_g.];
    my ($phrase_l)  = q[Hi!  I'm really_stupid's phrase_l.];

    print $phrase_g, "\n";
    print $phrase_l, "\n";
}
really_stupid;
print 'The value of $phrase_g is : ' . $phrase_g, "\n";
print 'The value of $phrase_l is : ' . $phrase_l, "\n";
```

- will print as its output:

```
Hi!  I'm really_stupid's phrase_g.
Hi!  I'm really_stupid's phrase_l.
The value of $phrase_g is : Hi!  I'm really_stupid's phrase_g.
The value of $phrase_l is :
```

One Final Subroutine Note

- We can call our subroutine simply by referencing its name:

```
really_stupid;
stupid_too;
```

- Prior to version 5 of Perl, subroutines were called with a leading '&' character, as follows:

```
&really_stupid;
&stupid_too;
```

- If you just can't get out of the C/C++/Java function mind set, you can also call subroutines this way:

```
really_stupid();
# &really_stupid(); is also okay.
stupid_too();
```

- They all mean *exactly the same thing*, and in your Perl travels you will see all of these uses - pick one, and stick to it

Exception Handling

- Here's a nice quote from Nigel Chapman's book:

```
"... everything is easy until something goes wrong."
```

- We have already seen the use of `die` to kill any running script, but, what if we want to catch errors and recover from them?

- Perl supports *exception handling*, which is a concept familiar to C++, Java, and Object Pascal programmers

- To do this, Perl provides the `eval` subroutine , which allows us to execute any piece of Perl code:

```
eval " print STDOUT qq[Hello world!\n]; ";
```

- will cause *another copy* of the Perl interpreter to load, and will pass the given Perl code to it for execution

More Exception Handling

- Interestingly, the following produces the same results:

```
$the_script = " print STDOUT qq[Hello world!\n]; ";
eval $the_script;
```

- Two things are important to note:
  - Under normal program control (i.e., within a Perl script), we can dynamically create a script for Perl to process
  - By using `eval` in this way, we are assured some protection from the Perl script in `$the_script` from causing problems

- Specifically, if a problem is detected, `eval` will return a undefined value, and will put an error code into the built-in variable `$@` (`$EVAL_ERROR` if we `'use English;'`)

- Most noteworthy is that fact that a call to `die`, inside the script passed to `eval`, *does not cause the death* of the script that called `eval` - we catch the exception instead, and an error message is put into the `$@` variable

Exception Handling Example

- The following script:

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

sub apollo_13 {
    die "Houston, we have a problem!";
}

print "Inside main script ... \n";

eval { apollo_13 };

print "Message from Apollo 13: $@" if $@;
print "Still inside main script.\n";
```

- will produce the following output on screen:

```
Inside main script ...
Message from Apollo 13: Houston, we have a problem! at test.pl line 3.
Still inside main script.
```