

Notes from Well House Consultants

These notes are written by Well House Consultants and distributed under their Open Training Notes License. If a copy of this license is not supplied at the end of these notes, please visit

*<http://www.wellho.net/net/whcotnl.html>
for details.*

1.1 Well House Consultants

Well House Consultants provides niche training, primarily but not exclusively in Open Source programming languages. We offer public courses at our training centre and private courses at your offices. We also make some of our training notes available under our "Open Training Notes" license, such as we're doing in this document here.

1.2 Open Training Notes License

With an "Open Training Notes License", for which we make no charge, you're allowed to print, use and distribute these notes provided that you retain the complete and unaltered license agreement with them, including our copyright statement. This means that you can learn from the notes, and have others learn from them too.

You are NOT allowed to charge (directly or indirectly) for the copying or distribution of these notes, nor are you allowed to charge for presentations making any use of them.

1.3 Courses presented by the author

If you would like us to attend a course (Java, Perl, Python, PHP, Tcl/Tk, MySQL or Linux) presented by the author of these notes, please see our public course schedule at

<http://www.wellho.net/course/index.html>

If you have a group of 4 or more trainees who require the same course at the same time, it will cost you less to have us run a private course for you. Please visit our onsite training page at

<http://www.wellho.net/course/otc.html>

which will give you details and costing information

1.4 Contact Details

Well House Consultants may be found online at

<http://www.wellho.net>

graham@wellho.net

technical contact

lisa@wellho.net

administration contact

Our full postal address is

404 The Spa
Melksham
Wiltshire
UK SN12 6QL

Phone +44 (0) 1225 708225

Fax +44 (0) 1225 707126

More Loops and Conditionals

"There's more than one way of doing it," says the cover of Programming Perl, the book co-written by Larry Wall, who wrote the language. He's right – there are usually many ways of doing the same thing. That's confusing for the newcomer, but great for the experienced practitioner. This module covers extra syntaxes for "if" and "while", and other loops and conditionals that use different keywords, or no keywords at all.

<i>The variety that is Perl</i>	4
<i>More conditional statements</i>	4
<i>More loop statements</i>	9
<i>Breaking a loop</i>	12
<i>Labels</i>	13
<i>The goto statement</i>	15
<i>Summary</i>	15

2.1 The variety that is Perl

Up to this point in the course, you've been using `if` statements for conditional text and `while` loops to repeat blocks of code. And you'll continue to use these; they're fundamental structures. But sometimes they're a bit wordy and there can be a better way of doing things.

In fact, there are so many ways to do these things in Perl that we'll show you examples of the most common ones and summarise some of the "well, if you must" type stuff.

Although we're talking about writing programs on this course, you may well find that a lot of your time is spent modifying, extending and repairing code that you've written earlier, or has been written by others. You should consider this maintenance aspect as you write your programs:

- Comment the programs well
- Design them in a modular way (structure or object oriented, a topic for later!)
- Have a version numbering system
- Write your Perl in such a way that it's easy for others to follow
- Set and use programming standards within your team or organisation.

These last two points mean that you should not try to use every different construct of the language!

I can't stress enough the importance of good coding practice in Perl. Perl is rather like a palette of paints, you can produce a fabulous picture, or you can produce a mess. A few people can just paint and produce art, but most of us need to discipline ourselves somewhat in order to produce a product that's fit for the job being done.

Here's a view of a customer of ours, a great fan of the Python language who works for a company that's a "Perl shop". See how he confirms my statements ;-)

"Secretly, I do like Perl but I 'think' in Python, and hate the non-maintainability that I see in Perl programs of just a few hundred lines of code. I think Perl users have a style for one-liners that doesn't scale to larger Perl programs. People need to write larger Perl programs with maintenance in mind." - Donald McCarthy

Because others will not have stuck to guidelines in the past, anyone who's responsible for maintaining code probably does need to have a good overview of all the facilities of the language, though.

2.2 More conditional statements

Here's a short program. You may remember something similar from when you first learnt about conditional statements:

```
#!/usr/bin/perl
# telegram - an if statement
print "please enter your age: ";
$age = <STDIN> ;
$togo = 100-$age;
if ( $age > 100 )
    {
        print "well done\n";
    } else {
        print "$togo years to your telegram\n";
    }
```

In England, when you reach the age of 100, you receive a telegram from The Queen – an old fashioned tradition that we celebrate in these examples ;-)

Important to note: The importance of good coding practice in Perl cannot be stressed enough.

Figure 1 Running Perl program
"telegram"

```
seal% telegram
please enter your age45
55 years to your telegram
seal% telegram
please enter your age102
well done
seal%
```

If -- single statement rather than a block

You've been writing

```
if ( $age > 100 )
{
    print "well done\n";
}
```

and if you've written C or Java in the past, you've been wishing you could leave out those braces. In Perl you can't, but you can write the `if` statement the other way around:

```
print "well done\n" if ($age > 100);
```

Good idea?

For: Saves coding. Makes it obvious it's a single statement block.

Against: Not intuitive. Conditional tends to get lost in body of code.

Unless -- an inverted if statement

You could replace

```
if ( $age > 100 )
{
    print "well done\n";
}
```

by

```
unless ($age <= 100)
{
    print "well done\n";
}
```

An `unless` statement is the same as an `if` statement, except that the block will be performed unless the condition is true. It's rather like an "if not". Yes, you could have also written

```
if (! ($age <= 100) )
{
    print "well done\n";
}
```

Just like `if`, you can add an `else` block after an `unless` statement. Just like `if`, you can write a single statement conditional the other way around:

```
print "well done\n" unless ($age <= 100);
```

Good idea?

For: Sometimes saves writing awkward negative conditions.

Against: Perhaps obscure for future maintainers

```
#!/usr/bin/perl
# tel2 - some if alternatives

print "please enter your age: ";
$age = <STDIN> ;
$togo = 100-$age;

print "Well done ... " if ( $age > 100 );
print "WELL done\n" unless ( $age <= 100 );
unless ( $age > 100 ) {
    print "$togo years to your telegram\n";
}
```

Figure 2 Running Perl program "tel2"

```
seal% tel2
please enter your age102
Well done ... WELL done
seal% tel2
please enter your age53
47 years to your telegram
seal%
```

Conditional operators

Consider the following statement:

```
if ( $age < 18 and $alcohol > 0 ) { print "no ...
```

You're checking to see if you can make a legal sale.

"If the age is less than 18, and the person has more than zero alcohol in his cart, reject the sale."

What if the person was aged 18 or over? Do you need to even look whether he has alcohol? No, it's legal anyway.

Perl knows this. It will only make the second check if necessary, in this case only if the purchaser is less than 18.

You can made use of this facility. Let's look at:

```
if ( $havebasket > 0 and ( $rdbasket = <STDIN> ) )
    { print "sale"; }
```

What does this do?

- Check and see if our customer has a basket
- If so, read basket contents and save to `$rdbasket`
- And if there was something in the basket, print "sale"

So we have the basket being read only if the customer has a basket. If we didn't want to print the word "sale" we could just have written

```
( $havebasket > 0 and ( $rdbasket = <STDIN> ) ) ;
```

or even

```
$havebasket > 0 and ( $rdbasket = <STDIN> ) ;
```

We now have conditional code – just the facility the `if` statement itself provides – without the need for the word `if`.

- You can read `and` as "and if true"
- You can read `or` as "and if false" or as "or"

Let's use them on the telegram example:

```
#!/usr/bin/perl
# tel3 - avoiding an if statement

print "please enter your age: ";
$age = <STDIN> ;
$togo = 100-$age;

$age > 100and print "well done\n";
$age > 100or print "$togo years to your telegram\n";
```

Figure 3 Running Perl program "tel3"

```
seal% tel3
please enter your age94
6 years to your telegram
seal%
```

You can use multiple **and** and **or** operators in a single statement; usual rules for brackets apply. Thus:

```
$havebasket > 0 and ($rdbasket = <STDIN>) and print "sale";
```

And there are some very common uses:

```
$age >= 18 or die "No sale. Customer too young\n";
```

Good idea?

For: Short, quick and clean

Against: Confusing to newbies. Limited to single statement.

As well as using the words **and** and **or**, Perl supports the **&&** and **||** operators. These operate on their operands in the same way, but they have a different level of precedence. In other words, you'll often need to add round brackets in different places if you switch between one and the other.

This script uses **and** and **&&** to run an assignment if a number entered by the user is not even.

```
#!/usr/bin/perl

### and, or, &&, || - precedence

print ("Enter a number: ");
chop ($num = <STDIN>);

$even = 1;
$num % 2 and ($even = 0);
print "Even value $even - test 1\n";

$even = 1;
$num % 2 && ($even = 0);
print "Even value $even - test 2\n";

$even = 1;
$num % 2 and $even = 0;
print "Even value $even - test 3\n";

__END__

$even = 1;
$num % 2 && $even = 0;
print "Even value $even - test 4\n"
```

If we run that, we get

```
[localhost:~/p1/p206] graham%andor
Enter a number:7
Even value 0 - test 1
Even value 0 - test 2
Even value 0 - test 3
[localhost:~/p1/p206] graham%
```

The fourth test has been commented out because it generates a syntax error. Here's the same program running without the `__END__` line:

```
[localhost:~/p1/p206] graham%andor
Can't modify logical and (&&) in scalar assignment at ./andor line 21, near "0;"
Execution of ./andor aborted due to compilation errors.
[localhost:~/p1/p206] graham%
```

A very odd message. In effect, the statement we were trying to compile was:

```
($num % 2 && $seven) = 0;
```

and the left hand side of the assignment isn't something that describes a variable's position in the computer memory, so it isn't a suitable target.

Note of caution: `&` and `|` perform a different operation to `&&` and `||`. Stick with `and` and `or`; not in vintage code (only arrived with Perl 5), but were long overdue!

The ?: operator

`&&` and `||` gave you shorthands for simple `if` and unless statements, but not a quick and easy way of writing `if else`. It's very common to write code such as

```
if ( $age > 100 )
{
  print "well done\n";
} else {
  print "$togo years to your telegram\n";
}
```

Perl provides an operator in three parts to help you with this.

```
(condition) ? (true action) : (false action)
```

so that you could have written

```
print ($age > 100) ? "well done\n" :
"$togo years to your telegram\n" ;
```

or even

```
($age > 100) ? print "well done\n":
  print "$togo years to your telegram\n";
```

Another very common use:

```
$largest = ($g>$h)?$g:$h;
```

and yet another:

```
print "You asked for ", $n, ($n!=1)
?" places\n": " place\n";
```

In use:

```
#!/usr/bin/perl
# tel4 - an if shorthand

print "please enter your age: ";
$age = <STDIN> ;
$togo = 100-$age;

( $age > 100 ) ?
    print "well done\n":
    print "$togo years to your telegram\n";

print "please enter number of places: ";
chop ($n = <STDIN>);
print "You asked for ",$n,($n!=1)?" places\n": " place\n"
```

Figure 4 Running Perl program "tel4"

```
seal% tel4
please enter your age34
66 years to your telegram
please enter number of places2
You asked for 2 places
seal% tel4
please enter your age101
well done
please enter number of places1
You asked for 1 place
seal%
```

Good idea?

For: Short, quick and clean

Against: Confusing to Perl newbies. Limited to single statement. ":" can get mistaken for ";" by the human reader

2.3 More loop statements

The **while** loop has, and will continue, to serve you well, but there are other loops too.

The until loop

Just as you could invert **if** by using **unless**, you can invert **while** using **until**.

Single statement while and until loops

Just as you can remove the { and } from a single statement **if** or **unless** (provided that you also write the condition on the end), so you can with **while** and **until**.

```
#!/usr/bin/perl
# power - next power of 2 above a number

print "Enter a number: ";
$num = <STDIN>;
$power = 1;
$power *=2 until ($power >= $num);
print "next power of 2 is $power\n";
```

```

seal% power
Enter a number:4
next power of 2 is 4
seal% power
Enter a number:100
next power of 2 is 128
seal% power
Enter a number:178
next power of 2 is 256
seal%

```

Figure 5 Running Perl program "power"

The for loop

Very frequently, you'll want to run a loop a certain number of times. Five working days in the week, or `$ndays` days on a course:

```

$now = 1;
while ($now <= 12) {
    print $now," times ",$number,
        " is ",$now*$number,"\n";
    $now++;
}

```

It works. But it's such a common requirement and the handling of the loop – the variable `$day` – is in three different statements. In this simple case they're all within five lines of code, but imagine a longer program in which they are a long way apart.

The `for` loop lets you take the three elements that control a loop such as this:

- Initial setting (s)
- Condition
- Action(s) before retesting the condition

and put all those elements into a single statement:

```

for ($now = 1;$now <= 12;$now++) {
    print $now," times ",$number,
        " is ",$now*$number,"\n";
}

```

The round brackets of a `for` loop must contain exactly two semicolons. The conditional section (between them) must give a true or false result. The first and third sections may be empty, or may even contain a comma separated list. It's possible to write some very confusing `for` loops. Keep them simple rather than trying to be too clever. The first loop in this example is good; the second may give you a headache!

```

#!/usr/bin/perl
# forloop - for loop

print "Enter a number = ";
chop ($number = <STDIN>);

for ($now = 1;$now <= 12;$now++) {
    print $now," times ",$number,
        " is ",$now*$number,"\n";
}

print "and also\n\n";

for (;$div < 12;
    print (++$div," divided by ",$number),
    print (" is ",$div/$number,"\n")){};

```

Figure 6 Running Perl program
"forloop"

```

seal% forloop
Enter a number =4
1 times 4 is 4
2 times 4 is 8
3 times 4 is 12
4 times 4 is 16
5 times 4 is 20
6 times 4 is 24
7 times 4 is 28
8 times 4 is 32
9 times 4 is 36
10 times 4 is 40
11 times 4 is 44
12 times 4 is 48
and also
1 divided by 4 is 0.25
2 divided by 4 is 0.5
3 divided by 4 is 0.75
4 divided by 4 is 1
5 divided by 4 is 1.25
6 divided by 4 is 1.5
7 divided by 4 is 1.75
8 divided by 4 is 2
9 divided by 4 is 2.25
10 divided by 4 is 2.5
11 divided by 4 is 2.75
12 divided by 4 is 3
seal%

```

The do - while loop

The **do - while** loop checks the condition at the end of the loop rather than at the beginning, thus allowing you to define a loop where the code runs at least once:

```

#!/usr/bin/perl

# a while loop that runs at least once

print ("Reads in names\n");

do {
    print ("Name: ");
    chop ($name = <STDIN>);
    $nst .= "$name ";
} while ($name =~ /\w/);

print ("Names were $nst\n");

```

When run, the program prompts for at least one name, even though the **\$name** variable does not match the regular expression when the loop is entered.

```

[localhost:~/pl/p206] graham%dowhile
Reads in names
Name:
Names were
[localhost:~/pl/p206] graham%dowhile
Reads in names

```

```
Name: Anne
Name: Bill
Name: Chloe
Name: David
Name:
Names were Anne Bill Chloe David
[localhost:~/p1/p206] graham%
```

You can also use a **do ... until** structure:

```
# A do ... until loop - reprompt until you get valid input.

$stop = 10;
do {
    print "Please enter a number over $stop: ";
    chop ($val = <STDIN>);
} until ($val > $stop);

print "Good. $val is over $stop\n";
```

Results:

```
$ perl dunt
Please enter a number over 10:
Please enter a number over 10:
Please enter a number over 10:11
Good. 11 is over 10
$
```

2.4 Breaking a loop

There will be times that you're in a loop, but as a result of testing a condition, you'll want to do one of these:

- Get out of the loop and carry on beyond
- Go around the loop again, with the same initial condition
- Move on to the next iteration of the loop

You can do each of these, using

```
last;   to leave the loop
redo;   to rerun with the same initial condition
next;   to move on to the next iteration
```

Here is an example using all three. This program asks the user to enter the number of course lunches required for each day of a five-day week.

- If the user enters a number over 12, an error is flagged and the user is prompted again – **redo;**
- If the user enters 0, the rest of that loop iteration is skipped as there's no need to run the code to place the order – **next;**
- If the user enters "end" then he's signalling that the course is finished and there are no more days to ask for – this is the **last;** time through the loop.

```
#!/usr/bin/perl
# jumps - loop

for ($day = 1; $day <=5; $day++) {
    print "Number of lunches, day $day: ";
    chop ($ndins = <STDIN>);

    $ndins =~ /^\

```

Figure 7 Running Perl program "jumps"

```
seal% jumps
Number of lunches, day 14
Ordering 4 dinners for day 1
Number of lunches, day 220
Too many. Try again
Number of lunches, day 22
Ordering 2 dinners for day 2
Number of lunches, day 30
Number of lunches, day 4end
Total dinners - 6
seal%
```

Although illustrated with a **for** loop, these statements can also be used with other loops such as **while** and **until**, and with the **foreach** loop we'll meet later.

2.5 Labels

"**last** gets you out of a loop". Yes, but which loop?

The normal answer is "the loop that you're in" but what if you're in a loop within a loop?

The exact rule is that "**last** gets you out of the innermost loop which you are in".

But what if that's not what you want? What if you want to get out of two nested loops?

- Label (name) the block you want to jump out of.
- Specify the label after the word **last**.

Labels must start at the beginning of a line and be the only thing on that line.

They do not follow the normal white space rules. The text of a label comprises a series of alphanumeric characters followed by a colon.

When you reference a label – in this example in the **last** statement – they follow the normal syntax rules. For example, they may appear anywhere on a line. The **:** should be omitted.

Next is an example (our course lunches again!).

If you enter **end** (lower case) it's the end of entry for this week. If you enter **END** (upper case) it's the end of the whole thing.

```
#!/usr/bin/perl
# jump2 - exit 2 loops

wjump:
for ($week =1; $week <=4; $week++) {

for ($day = 1; $day <=5; $day++) {
    print
    "Number of lunches, week $week, day $day: ";
        chop ($ndins = <STDIN>);

        $ndins =~ /^\

```

Figure 8 Running Perl program "jump2"

```
seal% jump2
Number of lunches, week 1, day 13
Number of lunches, week 1, day 24
Number of lunches, week 1, day 32
Number of lunches, week 1, day 4end
Dinners to date - 9
Number of lunches, week 2, day 13
Number of lunches, week 2, day 22
Number of lunches, week 2, day 39
Number of lunches, week 2, day 4END
Total dinners - 23
seal%
```

Perl has no switch or case statements, using a label and a block they're unnecessary:

```
#!/usr/bin/perl

while (1) {

print "please enter a word: ";
chop ($word = <STDIN>);
$co++;

choose:
{
($word =~ /quit/) and die "Leaving the program\n";
($word =~ /count/) and print ("count is $co\n") and last choose;
($word =~ /echo/) and print ("hello\n") and last choose;
print "eh?\n";
}

}
```

Running that, we get:

```
[localhost:~/pl/p206] graham%swit
please enter a word count
count is 1
please enter a word echo
hello
please enter a word count
count is 3
please enter a word shout
eh?
please enter a word quit
Leaving the program
[localhost:~/pl/p206] graham%
```

2.6 The goto statement

Get two programmers together, and suggest that "you don't need a goto statement" and you're likely to see a major argument break out. Some people find it very useful to be able to jump around their programs, but others say it makes the code very hard to test, hard to follow and hard to modify later.

But the philosophy of Perl is "if it might be useful, provide it". Syntax:

```
goto label
```

You may jump out of a block, but never into a block.

You'll learn later in this course that **goto** can do other things as well. You can get it to jump to a whole variety of different labels depending upon the value of some variable, and it does something which even the official texts call "highly magical" in certain other circumstances.

2.7 Summary

Should you only want to perform a single statement after an **if**, you can write it back-to-front and leave out the **{** and **}**. You can also replace **if** with **unless** to invert the condition.

An expression to the right of a **&&** (or the word **and**) is evaluated only if the condition to the left is true.

An expression to the right of a **||** (or the word **or**) is evaluated only if the condition to the left is false.

The **?...:** operator gives you a "shorthand" **if ... else** construct.

The **for** loop can be used as an elegant replacement for **while** loops.

```
for (initial; condition; subsequent) { ...
```

Within a loop, you can use

```
last    to exit the loop
```

```
redo    to rerun the current pass
```

```
next    to move on to the next pass
```

Normally, these affect your innermost loop but you can use labels so that they affect an outer block.

A label appears on a line on its own, followed by a colon.

The **goto** statement is available, but is poor practice and inefficient.

Exercise

Please try to do the following exercise *WITHOUT* using the words "while" or "if" in your program.

Write a program to ask the user to enter four numbers each between the value of 1 and 6. If the user enters a number below 1 or over 6, ask him to enter that number again. If the user enters the word `END`, stop reading numbers.

- Print out the count, average and total of all the numbers entered. Take care to print an appropriate error message for the average if no numbers were entered.

Our example answer is `dice`

```

graham@otter:~/profile/answers_ppdice
Value of die 1:5
Value of die 2:7
Invalid
Value of die 2:3
Value of die 3:1
Value of die 4:2
count: 4
total: 11
average: 2.75
graham@otter:~/profile/answers_ppdice
Value of die 1:1
Value of die 2:2
Value of die 3:4
Value of die 4:end
Invalid
Value of die 4:END
count: 3
total: 7
average: 2.333333333333333
graham@otter:~/profile/answers_ppdice
Value of die 1:END
count: 0
total: 0
average: infinity
graham@otter:~/profile/answers_pp>

```

License

*These notes are distributed under the **Well House Consultants Open Training Notes License**. Basically, if you distribute it and use it for free, we'll let you have it for free. If you charge for its distribution of use, we'll charge.*

3.1 Open Training Notes License

Training notes distributed under the **Well House Consultants Open Training Notes License** (WHCOTNL) may be reproduced for any purpose PROVIDE THAT:

- This License statement is retained, unaltered (save for additions to the change log) and complete.
- No charge is made for the distribution, nor for the use or application thereof. This means that you can use them to run training sessions or as support material for those sessions, but you cannot then make a charge for those training sessions.
- Alterations to the content of the document are clearly marked as being such, and a log of amendments is added below this notice.
- These notes are provided "as is" with no warranty of fitness for purpose. Whilst every attempt has been made to ensure their accuracy, no liability can be accepted for any errors of the consequences thereof.

Copyright is retained by Well House Consultants Ltd, of 404, The Spa, Melksham, Wiltshire, UK, SN12 6QL - phone number +44 (1) 1225 708225. Email contact - Graham Ellis (graham@wellho.net).

Please send any amendments and corrections to these notes to the Copyright holder - under the spirit of the Open Distribution license, we will incorporate suitable changes into future releases for the use of the community.

If you are charged for this material, or for presentation of a course (Other than by Well House Consultants) using this material, please let us know. It is a violation of the license under which this notes are distributed for such a charge to be made, except by the Copyright Holder.

If you would like Well House Consultants to use this material to present a training course for your organisation, or if you wish to attend a public course is one is available, please contact us or see our web site - <http://www.wellho.net> - for further details.

Change log
Original Version, Well House Consultants, 2004

Updated by: _____ on _____
Updated by: _____ on _____
Updated by: _____ on _____
Updated by: _____ on _____
Updated by: _____ on _____
Updated by: _____ on _____

License Ends.