Perl Short Course: Introduction

Duncan C. White (d.white@imperial.ac.uk)

Dept of Computing,
Imperial College London

December 2011

- This short practical course introduces you to Larry Wall's immensely flexible **Perl** programming language.

- This short practical course introduces you to Larry Wall's immensely flexible **Perl** programming language.
- It consists of a series of 6 one-hour lectures, with slides and practical examples.

- This short practical course introduces you to Larry Wall's immensely flexible **Perl** programming language.
- It consists of a series of 6 one-hour lectures, with slides and practical examples.
- The course materials for the course can be found at:

`http://www.doc.ic.ac.uk/~dcw/perl2011/`

- This short practical course introduces you to Larry Wall's immensely flexible **Perl** programming language.
- It consists of a series of 6 one-hour lectures, with slides and practical examples.
- The course materials for the course can be found at:

  `http://www.doc.ic.ac.uk/~dcw/perl2011/`

  - This first lecture will give a general introduction to Perl, a fast and shallow scan across most of the important features.

- This short practical course introduces you to Larry Wall's immensely flexible **Perl** programming language.
- It consists of a series of 6 one-hour lectures, with slides and practical examples.
- The course materials for the course can be found at:

`http://www.doc.ic.ac.uk/~dcw/perl2011/`

  - This first lecture will give a general introduction to Perl, a fast and shallow scan across most of the important features.
  - The second, third and fourth lectures will go over Perl in more detail, introducing most interesting features.

- This short practical course introduces you to Larry Wall's immensely flexible **Perl** programming language.
- It consists of a series of 6 one-hour lectures, with slides and practical examples.
- The course materials for the course can be found at:

  `http://www.doc.ic.ac.uk/~dcw/perl2011/`

  - This first lecture will give a general introduction to Perl, a fast and shallow scan across most of the important features.
  - The second, third and fourth lectures will go over Perl in more detail, introducing most interesting features.
  - The fifth and sixth lectures will describe the Perl module archive (CPAN) and some of it's modules, and how to construct your own modules and classes.

- This short practical course introduces you to Larry Wall's immensely flexible **Perl** programming language.
- It consists of a series of 6 one-hour lectures, with slides and practical examples.
- The course materials for the course can be found at:

  `http://www.doc.ic.ac.uk/~dcw/perl2011/`

  - This first lecture will give a general introduction to Perl, a fast and shallow scan across most of the important features.
  - The second, third and fourth lectures will go over Perl in more detail, introducing most interesting features.
  - The fifth and sixth lectures will describe the Perl module archive (CPAN) and some of it's modules, and how to construct your own modules and classes.
- There are two good books describing Perl: Randal Schwartz's excellent introduction **Learning Perl** and Larry Wall and Randal Schwartz's rather more advanced **Programming Perl.**

- Why yet another programming language? Why should I learn it?

- Why yet another programming language? Why should I learn it?
- We shall see that Perl was designed by a busy sysadmin - someone who needs a powerful language in which programs can be written, tested and deployed quickly to solve urgent problems.

- Why yet another programming language? Why should I learn it?
- We shall see that Perl was designed by a busy sysadmin - someone who needs a powerful language in which programs can be written, tested and deployed quickly to solve urgent problems.
- Such programs range from "Perl one liners" typed direct on the command line as part of a pipeline, to properly designed, engineered and maintained large modular programs.

- Why yet another programming language? Why should I learn it?
- We shall see that Perl was designed by a busy sysadmin - someone who needs a powerful language in which programs can be written, tested and deployed quickly to solve urgent problems.
- Such programs range from "Perl one liners" typed direct on the command line as part of a pipeline, to properly designed, engineered and maintained large modular programs.
- Perl gives us great **leverage** in writing programs when compared to languages like C, C++ or Java. Instead of starting from the architectural level (highly machine-efficient but not programmer-efficient) Perl lets you program at a level much higher up, with more powerful tools.

- Why yet another programming language? Why should I learn it?
- We shall see that Perl was designed by a busy sysadmin - someone who needs a powerful language in which programs can be written, tested and deployed quickly to solve urgent problems.
- Such programs range from "Perl one liners" typed direct on the command line as part of a pipeline, to properly designed, engineered and maintained large modular programs.
- Perl gives us great **leverage** in writing programs when compared to languages like C, C++ or Java. Instead of starting from the architectural level (highly machine-efficient but not programmer-efficient) Perl lets you program at a level much higher up, with more powerful tools.
- Perl is an immensely pragmatic language, borrowing the best features from many other languages - forming a coherent whole, more powerful than the sum of its parts.

- Why yet another programming language? Why should I learn it?
- We shall see that Perl was designed by a busy sysadmin - someone who needs a powerful language in which programs can be written, tested and deployed quickly to solve urgent problems.
- Such programs range from "Perl one liners" typed direct on the command line as part of a pipeline, to properly designed, engineered and maintained large modular programs.
- Perl gives us great **leverage** in writing programs when compared to languages like C, C++ or Java. Instead of starting from the architectural level (highly machine-efficient but not programmer-efficient) Perl lets you program at a level much higher up, with more powerful tools.
- Perl is an immensely pragmatic language, borrowing the best features from many other languages - forming a coherent whole, more powerful than the sum of its parts.
- Perl is known as the **Swiss Army Chainsaw** of programming; it makes *the easy tasks easy, the hard tasks possible*.

Perl borrows the best from several different languages, and binds them together into a seamless whole:

Perl borrows the best from several different languages, and binds them together into a seamless whole:

- Perl's **control structures** come from C and the shell.

Perl borrows the best from several different languages, and binds them together into a seamless whole:

- Perl's **control structures** come from C and the shell.
- Perl's **expression syntax** comes from C, with several operators brought in from the shell – such as a set of file test operators.

Perl borrows the best from several different languages, and binds them together into a seamless whole:

- Perl's **control structures** come from C and the shell.
- Perl's **expression syntax** comes from C, with several operators brought in from the shell – such as a set of file test operators.
- Perl provides more **powerful data types** (dynamic **arrays, lists** and **hashes** - associative arrays) than most other languages, and makes them very easy to use!

Perl borrows the best from several different languages, and binds them together into a seamless whole:

- Perl's **control structures** come from C and the shell.
- Perl's **expression syntax** comes from C, with several operators brought in from the shell – such as a set of file test operators.
- Perl provides more **powerful data types** (dynamic **arrays, lists** and **hashes** - associative arrays) than most other languages, and makes them very easy to use!
- Perl has **regular expressions** built-in (as used in filters **sed** and **grep**), and extends them in a gradually increasing number of ways to make them even more powerful.

Perl borrows the best from several different languages, and binds them together into a seamless whole:

- Perl's **control structures** come from C and the shell.
- Perl's **expression syntax** comes from C, with several operators brought in from the shell – such as a set of file test operators.
- Perl provides more **powerful data types** (dynamic **arrays, lists** and **hashes** - associative arrays) than most other languages, and makes them very easy to use!
- Perl has **regular expressions** built-in (as used in filters **sed** and **grep**), and extends them in a gradually increasing number of ways to make them even more powerful.
- Perl gives you the ability to **build filters** easily - to **manipulate files, processes** and **command line arguments** simply and efficiently.

Perl borrows the best from several different languages, and binds them together into a seamless whole:

- Perl's **control structures** come from C and the shell.
- Perl's **expression syntax** comes from C, with several operators brought in from the shell – such as a set of file test operators.
- Perl provides more **powerful data types** (dynamic **arrays, lists** and **hashes** - associative arrays) than most other languages, and makes them very easy to use!
- Perl has **regular expressions** built-in (as used in filters **sed** and **grep**), and extends them in a gradually increasing number of ways to make them even more powerful.
- Perl gives you the ability to **build filters** easily - to **manipulate files, processes** and **command line arguments** simply and efficiently.
- Most crucially, Perl does all **storage management** for us - just like **awk,** and **Java**, unlike **C**.

Perl borrows the best from several different languages, and binds them together into a seamless whole:

- Perl's **control structures** come from C and the shell.
- Perl's **expression syntax** comes from C, with several operators brought in from the shell – such as a set of file test operators.
- Perl provides more **powerful data types** (dynamic **arrays, lists** and **hashes** - associative arrays) than most other languages, and makes them very easy to use!
- Perl has **regular expressions** built-in (as used in filters **sed** and **grep**), and extends them in a gradually increasing number of ways to make them even more powerful.
- Perl gives you the ability to **build filters** easily - to **manipulate files, processes** and **command line arguments** simply and efficiently.
- Most crucially, Perl does all **storage management** for us - just like **awk,** and **Java**, unlike **C**.
- Plus: threads, portable graphics, OOP, functional programming, network programming and more modules than you can count.

The main ways Perl does storage management are as follows:

The main ways Perl does storage management are as follows:

- **Strings are a basic type** - not arrays of characters as they are in C. Strings **grow as needed** - and they can be enormous!

The main ways Perl does storage management are as follows:

- **Strings are a basic type** - not arrays of characters as they are in C. Strings **grow as needed** - and they can be enormous!
- **Arrays grow as needed** - simply assign to an element and the array extends to include that element automatically. All elements that have not had a value assigned to them have the undefined value, effectively zero.

The main ways Perl does storage management are as follows:

- **Strings are a basic type** - not arrays of characters as they are in C. Strings **grow as needed** - and they can be enormous!
- **Arrays grow as needed** - simply assign to an element and the array extends to include that element automatically. All elements that have not had a value assigned to them have the undefined value, effectively zero.
- Perl provides **unlimited length Prolog-style lists**, actually arrays. Perl provides many powerful list/array operators.

The main ways Perl does storage management are as follows:

- **Strings are a basic type** - not arrays of characters as they are in C. Strings **grow as needed** - and they can be enormous!
- **Arrays grow as needed** - simply assign to an element and the array extends to include that element automatically. All elements that have not had a value assigned to them have the undefined value, effectively zero.
- Perl provides **unlimited length Prolog-style lists**, actually arrays. Perl provides many powerful list/array operators.
- These lists can also act as **Prolog-style tuples** - an unnamed collection of data like a record.

The main ways Perl does storage management are as follows:

- **Strings are a basic type** - not arrays of characters as they are in C. Strings **grow as needed** - and they can be enormous!
- **Arrays grow as needed** - simply assign to an element and the array extends to include that element automatically. All elements that have not had a value assigned to them have the undefined value, effectively zero.
- Perl provides **unlimited length Prolog-style lists**, actually arrays. Perl provides many powerful list/array operators.
- These lists can also act as **Prolog-style tuples** - an unnamed collection of data like a record.
- Perl also provides **associative arrays**, ie. arrays where the index is an arbitrary string - i.e. a collection of (key,value) pairs indexed by key. These are called **hashes** in Perl speak.

The main ways Perl does storage management are as follows:

- **Strings are a basic type** - not arrays of characters as they are in C. Strings **grow as needed** - and they can be enormous!
- **Arrays grow as needed** - simply assign to an element and the array extends to include that element automatically. All elements that have not had a value assigned to them have the undefined value, effectively zero.
- Perl provides **unlimited length Prolog-style lists**, actually arrays. Perl provides many powerful list/array operators.
- These lists can also act as **Prolog-style tuples** - an unnamed collection of data like a record.
- Perl also provides **associative arrays**, ie. arrays where the index is an arbitrary string - i.e. a collection of (key,value) pairs indexed by key. These are called **hashes** in Perl speak.
- To do anything more complex, eg. multi-dimensional arrays, Perl provides **references** - an ability for one variable to refer to another variable. Rather like pointers.

- Ok, let's jump in and get started. Following a long tradition, our first Perl program will be the classic "hello world" program.

- Ok, let's jump in and get started. Following a long tradition, our first Perl program will be the classic "hello world" program.
- Using any editor (vi, pico, nedit, emacs), we create a file called **eg1**, containing the following lines:

```
#
# eg1: a first Perl program
#
print "hello world\n";
```

- Ok, let's jump in and get started. Following a long tradition, our first Perl program will be the classic "hello world" program.
- Using any editor (vi, pico, nedit, emacs), we create a file called **eg1**, containing the following lines:

```
#
# eg1: a first Perl program
#
print "hello world\n";
```

- Then, we syntax check the program:

```
perl -cw eg1
```

- Ok, let's jump in and get started. Following a long tradition, our first Perl program will be the classic "hello world" program.
- Using any editor (vi, pico, nedit, emacs), we create a file called **eg1**, containing the following lines:

```
#
# eg1: a first Perl program
#
print "hello world\n";
```

- Then, we syntax check the program:

```
perl -cw eg1
```

- Finally, we run the program by typing:

```
perl eg1
```

- Ok, let's jump in and get started. Following a long tradition, our first Perl program will be the classic "hello world" program.
- Using any editor (vi, pico, nedit, emacs), we create a file called **eg1**, containing the following lines:

  ```
  #
  # eg1: a first Perl program
  #
  print "hello world\n";
  ```
- Then, we syntax check the program:

  ```
  perl -cw eg1
  ```
- Finally, we run the program by typing:

  ```
  perl eg1
  ```
- What can we see immediately about Perl from this example?
  - Lines beginning with # are comments, ignored by Perl.

- Ok, let's jump in and get started. Following a long tradition, our first Perl program will be the classic "hello world" program.
- Using any editor (vi, pico, nedit, emacs), we create a file called **eg1**, containing the following lines:

```
#
# eg1: a first Perl program
#
print "hello world\n";
```

- Then, we syntax check the program:

```
perl -cw eg1
```

- Finally, we run the program by typing:

```
perl eg1
```

- What can we see immediately about Perl from this example?
  - Lines beginning with # are comments, ignored by Perl.
  - Statement are terminated with semi-colons.

- Ok, let's jump in and get started. Following a long tradition, our first Perl program will be the classic "hello world" program.
- Using any editor (vi, pico, nedit, emacs), we create a file called **eg1**, containing the following lines:

```
#
# eg1: a first Perl program
#
print "hello world\n";
```

- Then, we syntax check the program:

```
perl -cw eg1
```

- Finally, we run the program by typing:

```
perl eg1
```

- What can we see immediately about Perl from this example?
  - Lines beginning with # are comments, ignored by Perl.
  - Statement are terminated with semi-colons.
  - A string is placed in double quotes, and can contain C-style special characters such as `\n`.

- Suppose we now create eg2 containing the following:

```
print "Please enter your name: ";
my $name = <STDIN>;
print "\nhello $name!\n";
```

- What's going on here?

- Suppose we now create eg2 containing the following:

```
print "Please enter your name: ";
my $name = <STDIN>;
print "\nhello $name!\n";
```

- What's going on here?
  - `my $name` declares a *scalar variable*. It can hold a number (integer or real) or an arbitrary length string. The $ is always present.

- Suppose we now create eg2 containing the following:

```
print "Please enter your name: ";
my $name = <STDIN>;
print "\nhello $name!\n";
```

- What's going on here?
  - `my $name` declares a *scalar variable*. It can hold a number (integer or real) or an arbitrary length string. The $ is always present.
  - `<STDIN>` means *read one line from stdin*.

Personalising the Message (eg2)

- Suppose we now create eg2 containing the following:

```
print "Please enter your name: ";
my $name = <STDIN>;
print "\nhello $name!\n";
```

- What's going on here?
  - `my $name` declares a *scalar variable*. It can hold a number (integer or real) or an arbitrary length string. The $ is always present.
  - `<STDIN>` means *read one line from stdin*.
  - The line read is then assigned to `$name`.

- Suppose we now create eg2 containing the following:

```
print "Please enter your name: ";
my $name = <STDIN>;
print "\nhello $name!\n";
```

- What's going on here?
  - `my $name` declares a *scalar variable*. It can hold a number (integer or real) or an arbitrary length string. The $ is always present.
  - `<STDIN>` means *read one line from stdin*.
  - The line read is then assigned to `$name`.
  - The second print statement prints the result of the string `"\nhello $name!\n"` after *variable interpolation*: the current value of `$name` is interpolated into the string in place of the character sequence `$name`.

- Suppose we now create eg2 containing the following:

```
print "Please enter your name: ";
my $name = <STDIN>;
print "\nhello $name!\n";
```

- What's going on here?
  - `my $name` declares a *scalar variable*. It can hold a number (integer or real) or an arbitrary length string. The $ is always present.
  - `<STDIN>` means *read one line from stdin*.
  - The line read is then assigned to `$name`.
  - The second print statement prints the result of the string `"\nhello $name!\n"` after *variable interpolation*: the current value of `$name` is interpolated into the string in place of the character sequence `$name`.
  - For instance, if `$name = "duncan"` then the string would be `"\nhello duncan!\n"`.

- Suppose we now create eg2 containing the following:

```
print "Please enter your name: ";
my $name = <STDIN>;
print "\nhello $name!\n";
```

- What's going on here?
  - `my $name` declares a *scalar variable*. It can hold a number (integer or real) or an arbitrary length string. The $ is always present.
  - `<STDIN>` means *read one line from stdin*.
  - The line read is then assigned to `$name`.
  - The second print statement prints the result of the string `"\nhello $name!\n"` after *variable interpolation*: the current value of `$name` is interpolated into the string in place of the character sequence `$name`.
  - For instance, if `$name = "duncan"` then the string would be `"\nhello duncan!\n"`.
- Once again, we syntax check eg2 and then run it.

- Suppose we now create eg2 containing the following:

```
print "Please enter your name: ";
my $name = <STDIN>;
print "\nhello $name!\n";
```

- What's going on here?
  - `my $name` declares a *scalar variable*. It can hold a number (integer or real) or an arbitrary length string. The $ is always present.
  - `<STDIN>` means *read one line from stdin*.
  - The line read is then assigned to `$name`.
  - The second print statement prints the result of the string `"\nhello $name!\n"` after *variable interpolation*: the current value of `$name` is interpolated into the string in place of the character sequence `$name`.
  - For instance, if `$name = "duncan"` then the string would be `"\nhello duncan!\n"`.
- Once again, we syntax check eg2 and then run it.
- Was there anything that surprised you when the program ran?

- Because the <> operator reads a line and returns it *including the newline at the end*.

- Because the <> operator reads a line and returns it *including the newline at the end*.
- To get round this common problem, add:

```
chomp $name;
```

  immediately after reading `$name`. This deletes a trailing newline.

- Because the <> operator reads a line and returns it *including the newline at the end*.
- To get round this common problem, add:

  ```
  chomp $name;
  ```

  immediately after reading `$name`. This deletes a trailing newline.
- Rerun and check that the `!` is now on the same line as the name.

- Because the <> operator reads a line and returns it *including the newline at the end*.
- To get round this common problem, add:

  ```
  chomp $name;
  ```

  immediately after reading `$name`. This deletes a trailing newline.
- Rerun and check that the `!` is now on the same line as the name.
- Suppose we now wish to lowercase the whole name, and then capitalise the first letter:

- Because the <> operator reads a line and returns it *including the newline at the end*.
- To get round this common problem, add:

  ```
          chomp $name;
  ```

  immediately after reading `$name`. This deletes a trailing newline.
- Rerun and check that the `!` is now on the same line as the name.
- Suppose we now wish to lowercase the whole name, and then capitalise the first letter:
- We simply add (after the `chomp`):

  ```
  $name = ucfirst(lc($name));
  ```

- Because the <> operator reads a line and returns it *including the newline at the end*.
- To get round this common problem, add:

  ```
          chomp $name;
  ```

  immediately after reading `$name`. This deletes a trailing newline.
- Rerun and check that the `!` is now on the same line as the name.
- Suppose we now wish to lowercase the whole name, and then capitalise the first letter:
- We simply add (after the `chomp`):

  ```
  $name = ucfirst(lc($name));
  ```
- Recheck it and rerun it now.

- Because the <> operator reads a line and returns it *including the newline at the end*.
- To get round this common problem, add:

```
        chomp $name;
```

  immediately after reading `$name`. This deletes a trailing newline.
- Rerun and check that the `!` is now on the same line as the name.
- Suppose we now wish to lowercase the whole name, and then capitalise the first letter:
- We simply add (after the `chomp`):

```
$name = ucfirst(lc($name));
```

- Recheck it and rerun it now.
- How long would that have taken to write in C *and know it's bug-free*?

- Suppose we now want a special-case - if the name is your first name, print out a special message:

- Suppose we now want a special-case - if the name is your first name, print out a special message:
- Embed the final print inside the else part of the following new if statement:

```
if( $name eq "Duncan" )
{
        print "\nwotcha Dunc mate!\n";
} else
{
        print "\nhello $name!\n";
}
```

- Suppose we now want a special-case - if the name is your first name, print out a special message:
- Embed the final print inside the else part of the following new if statement:

```
if( $name eq "Duncan" )
{
        print "\nwotcha Dunc mate!\n";
} else
{
        print "\nhello $name!\n";
}
```

- Syntax check, run it again a few times. Check it works.
- You may wish to try this with your own name instead..

- Let's just refresh our memories - the complete program is now:

```
#
# eg4: special case greeting
#
print "Please enter your name: ";
my $name = <STDIN>;
chomp $name;
$name = ucfirst(lc($name));
if( $name eq "Duncan" )
{
        print "\nwotcha Dunc mate!\n";
} else
{
        print "\nhello $name!\n";
}
```

- So far, the name entered (after capitalization changes) must be your first name *exactly* for the special case to apply.

- So far, the name entered (after capitalization changes) must be your first name *exactly* for the special case to apply.
- For example, `duncan`, `DUNCAN`, `Duncan`, `DunCAn` are accepted, but `Dunc`, `Dunk` and the regrettable `Dunky Babe` are not.

- So far, the name entered (after capitalization changes) must be your first name *exactly* for the special case to apply.
- For example, `duncan`, `DUNCAN`, `Duncan`, `DunCAn` are accepted, but `Dunc`, `Dunk` and the regrettable `Dunky Babe` are not.
- Replace the **if** condition with:

  ```
  if( $name =~ /^Dun[ck]/ )
  ```

  What on earth does this mean?

- So far, the name entered (after capitalization changes) must be your first name *exactly* for the special case to apply.
- For example, `duncan`, `DUNCAN`, `Duncan`, `DunCAn` are accepted, but `Dunc`, `Dunk` and the regrettable `Dunky Babe` are not.
- Replace the **if** condition with:

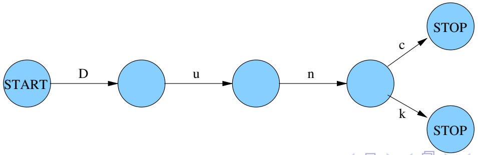
  ```
  if( $name =~ /^Dun[ck]/ )
  ```

  What on earth does this mean?
- This is an example of matching a string against a *regular expression* - known as a *regex* - as found in the Unix filters **sed, grep** and **awk**.

- So far, the name entered (after capitalization changes) must be your first name *exactly* for the special case to apply.
- For example, `duncan`, `DUNCAN`, `Duncan`, `DunCAn` are accepted, but `Dunc`, `Dunk` and the regrettable `Dunky Babe` are not.
- Replace the **if** condition with:

  ```
  if( $name =~ /^Dun[ck]/ )
  ```

  What on earth does this mean?
- This is an example of matching a string against a *regular expression* - known as a *regex* - as found in the Unix filters **sed, grep** and **awk**.
- **regexes** are explained in more detail later, so for now let's just say that it succeeds if `$name` starts with the string `"Dun"`, immediately followed by *either* `"c"` or `"k"`. Graphically:



- Now, suppose that we require a secret word from anyone other than "Dunc". Right at the top, below the comment lines, add:

```
my $secretword = "Klingon";
```

- Now, suppose that we require a secret word from anyone other than "Dunc". Right at the top, below the comment lines, add:

  ```
  my $secretword = "Klingon";
  ```

- In the **else** part, add the following:

  ```
  print "What is the secret word: ";
  while(1)
  {
          my $guess = <STDIN>;
          chomp $guess;
  last if $guess eq $secretword;
          print "Wrong - guess again: ";
  }
  ```

- Now, suppose that we require a secret word from anyone other than "Dunc". Right at the top, below the comment lines, add:

```
my $secretword = "Klingon";
```

- In the **else** part, add the following:

```
print "What is the secret word: ";
while(1)
{
        my $guess = <STDIN>;
        chomp $guess;
last if $guess eq $secretword;
        print "Wrong - guess again: ";
}
```

- This is an infinite **while** loop. Inside, we obtain a line of input, store it in $guess and chomp it as usual.

- Now, suppose that we require a secret word from anyone other than "Dunc". Right at the top, below the comment lines, add:

```
my $secretword = "Klingon";
```

- In the **else** part, add the following:

```
print "What is the secret word: ";
while(1)
{
        my $guess = <STDIN>;
        chomp $guess;
last if $guess eq $secretword;
        print "Wrong - guess again: ";
}
```

- This is an infinite **while** loop. Inside, we obtain a line of input, store it in $guess and chomp it as usual.
- We break out of the loop (**last**) if the guess is exactly the same as the secret word. If the guess is wrong, we continue round the loop. (Notice the **last if** alternative syntax).

- Suppose we now want several secret words. We want a **list**!

- Suppose we now want several secret words. We want a **list**!
- Replace the assignment to `$secretword` at the top with:

```
my @secretword = ( "Klingon", "Romulan", "Vulcan" );
```

- Suppose we now want several secret words. We want a **list**!
- Replace the assignment to `$secretword` at the top with:

```
my @secretword = ( "Klingon", "Romulan", "Vulcan" );
```

- Replace the **last if** line with:

```
my $correct = 0;
foreach my $i (@secretword)
{
        $correct = 1 if $i eq $guess;
}
last if $correct;
```

- Suppose we now want several secret words. We want a **list**!
- Replace the assignment to $secretword at the top with:

```
my @secretword = ( "Klingon", "Romulan", "Vulcan" );
```

- Replace the **last if** line with:

```
my $correct = 0;
foreach my $i (@secretword)
{
        $correct = 1 if $i eq $guess;
}
last if $correct;
```

- Now any of the secret words will be accepted.

- Suppose we now want several secret words. We want a **list**!
- Replace the assignment to `$secretword` at the top with:

```
my @secretword = ( "Klingon", "Romulan", "Vulcan" );
```

- Replace the **last if** line with:

```
my $correct = 0;
foreach my $i (@secretword)
{
        $correct = 1 if $i eq $guess;
}
last if $correct;
```

- Now any of the secret words will be accepted.
- But there's something slightly strange about what we just did: We stored the words in an ordered list and sequenced through it.

- Suppose we now want several secret words. We want a **list**!
- Replace the assignment to `$secretword` at the top with:

```
my @secretword = ( "Klingon", "Romulan", "Vulcan" );
```

- Replace the **last if** line with:

```
my $correct = 0;
foreach my $i (@secretword)
{
        $correct = 1 if $i eq $guess;
}
last if $correct;
```

- Now any of the secret words will be accepted.
- But there's something slightly strange about what we just did: We stored the words in an ordered list and sequenced through it.
- But we didn't want to sequence through a list: We wanted a *set of secret words* and to test *set membership*.

- A Perl *hash* stores an arbitrary number of **(key, value)** pairs, indexing the keys.

- A Perl *hash* stores an arbitrary number of **(key, value)** pairs, indexing the keys.
- Replace the `@secretword` initialization with:

```
my %issecretword = ();
$issecretword{"Klingon"} = 1;
$issecretword{"Romulan"} = 1;
$issecretword{"Vulcan"} = 1;
```

- A Perl *hash* stores an arbitrary number of **(key, value)** pairs, indexing the keys.
- Replace the `@secretword` initialization with:

```
my %issecretword = ();
$issecretword{"Klingon"} = 1;
$issecretword{"Romulan"} = 1;
$issecretword{"Vulcan"} = 1;
```

- Now replace the entire `$correct` loop we just added with:

```
last if $issecretword{$guess};
```

- A Perl *hash* stores an arbitrary number of **(key, value)** pairs, indexing the keys.
- Replace the `@secretword` initialization with:

```
my %issecretword = ();
$issecretword{"Klingon"} = 1;
$issecretword{"Romulan"} = 1;
$issecretword{"Vulcan"} = 1;
```

- Now replace the entire `$correct` loop we just added with:

```
last if $issecretword{$guess};
```

- Here we use a **hash** (where all the values are 1) as a **set**.

- A Perl *hash* stores an arbitrary number of **(key, value)** pairs, indexing the keys.
- Replace the `@secretword` initialization with:

```
my %issecretword = ();
$issecretword{"Klingon"} = 1;
$issecretword{"Romulan"} = 1;
$issecretword{"Vulcan"} = 1;
```

- Now replace the entire `$correct` loop we just added with:

```
last if $issecretword{$guess};
```

- Here we use a **hash** (where all the values are 1) as a **set**.
- Hashes take quite a bit of getting used to! Not many languages support them - in C, you'd probably have used an array and linear search.

- A Perl *hash* stores an arbitrary number of **(key, value)** pairs, indexing the keys.
- Replace the `@secretword` initialization with:

```
my %issecretword = ();
$issecretword{"Klingon"} = 1;
$issecretword{"Romulan"} = 1;
$issecretword{"Vulcan"} = 1;
```

- Now replace the entire `$correct` loop we just added with:

```
last if $issecretword{$guess};
```

- Here we use a **hash** (where all the values are 1) as a **set**.
- Hashes take quite a bit of getting used to! Not many languages support them - in C, you'd probably have used an array and linear search.
- Once you've got used to hashes, you never want to be without them! Many data structures you would build using pointers etc in C can be done with the combination of lists and hashes.

- It's a bit stupid to store the secret words in plain view inside the Perl script. Let's store them in plain view in a text file instead:-)

- It's a bit stupid to store the secret words in plain view inside the Perl script. Let's store them in plain view in a text file instead:-)
- Near the top, add:

```
use IO::File;
```

- It's a bit stupid to store the secret words in plain view inside the Perl script. Let's store them in plain view in a text file instead:-)
- Near the top, add:

```
use IO::File;
```

- Replace the initialisation of the `$issecretword` elements with:

```
my $in = new IO::File( "secretwords" ) || die;
while( my $line = <$in> )
{
        chomp $line;
        $issecretword{$line} = 1;
}
$in->close;
```

- It's a bit stupid to store the secret words in plain view inside the Perl script. Let's store them in plain view in a text file instead:-)
- Near the top, add:

```
use IO::File;
```

- Replace the initialisation of the `$issecretword` elements with:

```
my $in = new IO::File( "secretwords" ) || die;
while( my $line = <$in> )
{
        chomp $line;
        $issecretword{$line} = 1;
}
$in->close;
```

- This shows us Perl's classic idiom *foreach line in a file*:
    - We open a text file called `secretwords`, exiting if the open fails.

- It's a bit stupid to store the secret words in plain view inside the Perl script. Let's store them in plain view in a text file instead:-)
- Near the top, add:

```
use IO::File;
```

- Replace the initialisation of the `$issecretword` elements with:

```
my $in = new IO::File( "secretwords" ) || die;
while( my $line = <$in> )
{
        chomp $line;
        $issecretword{$line} = 1;
}
$in->close;
```

- This shows us Perl's classic idiom *foreach line in a file*:
    - We open a text file called `secretwords`, exiting if the open fails.
    - While the file contains another line of text, read it.

- It's a bit stupid to store the secret words in plain view inside the Perl script. Let's store them in plain view in a text file instead:-)
- Near the top, add:

```
use IO::File;
```

- Replace the initialisation of the `$issecretword` elements with:

```
my $in = new IO::File( "secretwords" ) || die;
while( my $line = <$in> )
{
        chomp $line;
        $issecretword{$line} = 1;
}
$in->close;
```

- This shows us Perl's classic idiom *foreach line in a file*:
  - We open a text file called `secretwords`, exiting if the open fails.
  - While the file contains another line of text, read it.
  - Then we process the line - in this case, add the line to the sethash.

- It's a bit stupid to store the secret words in plain view inside the Perl script. Let's store them in plain view in a text file instead:-)
- Near the top, add:

```
use IO::File;
```

- Replace the initialisation of the `$issecretword` elements with:

```
my $in = new IO::File( "secretwords" ) || die;
while( my $line = <$in> )
{
        chomp $line;
        $issecretword{$line} = 1;
}
$in->close;
```

- This shows us Perl's classic idiom *foreach line in a file*:
    - We open a text file called `secretwords`, exiting if the open fails.
    - While the file contains another line of text, read it.
    - Then we process the line - in this case, add the line to the sethash.
    - When we have read the last line from the file, quit the **while** loop and close the file.

- There is an even better way of storing the secret words on disk and retrieving them: Unix provides a highly efficient storage system called DBM that stores arbitrary **(key,value)** string pairs.

- There is an even better way of storing the secret words on disk and retrieving them: Unix provides a highly efficient storage system called DBM that stores arbitrary **(key,value)** string pairs.
- Sound familiar? Perl provides a trivial interface to DBM files - the concepts of DBM map perfectly onto a Perl hash.

- There is an even better way of storing the secret words on disk and retrieving them: Unix provides a highly efficient storage system called DBM that stores arbitrary **(key,value)** string pairs.
- Sound familiar? Perl provides a trivial interface to DBM files - the concepts of DBM map perfectly onto a Perl hash.
- Now, for the first time, we need two programs: one to initialise the DBM file, and our existing program (modified a bit) to read the DBM file:

- There is an even better way of storing the secret words on disk and retrieving them: Unix provides a highly efficient storage system called DBM that stores arbitrary **(key,value)** string pairs.
- Sound familiar? Perl provides a trivial interface to DBM files - the concepts of DBM map perfectly onto a Perl hash.
- Now, for the first time, we need two programs: one to initialise the DBM file, and our existing program (modified a bit) to read the DBM file:
- First, the creation program **mksecret** is as follows:

```
#
#       mksecret: create the secret words DBM file
#
dbmopen( my %secret, "secretwords", 0666 ) || die;
$secret{"Romulan"} = 1;
$secret{"Klingon"} = 1;
$secret{"Vulcan"} = 1;
dbmclose( %secret );
```

- Back in the main program - now called **eg10** - remove all the file reading code (up to `CLOSE(IN)`) and replace it with:

```
dbmopen(my %issecretword, "secretwords", 0666) || die;
```

- Back in the main program - now called **eg10** - remove all the file reading code (up to `CLOSE(IN)`) and replace it with:

```
dbmopen(my %issecretword, "secretwords", 0666) || die;
```

- Right at the end of the program, add:

```
dbmclose( %issecretword );
```

- Back in the main program - now called **eg10** - remove all the file reading code (up to `CLOSE(IN)`) and replace it with:

```
dbmopen(my %issecretword, "secretwords", 0666) || die;
```

- Right at the end of the program, add:

```
dbmclose( %issecretword );
```

- Now, the file reading is done entirely automatically for you.

- Back in the main program - now called **eg10** - remove all the file reading code (up to `CLOSE(IN)`) and replace it with:

```
dbmopen(my %issecretword, "secretwords", 0666) || die;
```

- Right at the end of the program, add:

```
dbmclose( %issecretword );
```

- Now, the file reading is done entirely automatically for you.
- But more efficiently: the DBM file is not a plain text file. In a large DBM file containing millions of **(key, value)** pairs, retrieving the value corresponding to a specific key is usually done using **only two disk accesses**!

- Back in the main program - now called **eg10** - remove all the file reading code (up to `CLOSE(IN)`) and replace it with:

```
dbmopen(my %issecretword, "secretwords", 0666) || die;
```

- Right at the end of the program, add:

```
dbmclose( %issecretword );
```

- Now, the file reading is done entirely automatically for you.
- But more efficiently: the DBM file is not a plain text file. In a large DBM file containing millions of **(key, value)** pairs, retrieving the value corresponding to a specific key is usually done using **only two disk accesses**!
- By using this highly efficient system, we get *persistent storage* for Perl programs, for free!

- Back in the main program - now called **eg10** - remove all the file reading code (up to `CLOSE(IN)`) and replace it with:

```
dbmopen(my %issecretword, "secretwords", 0666) || die;
```

- Right at the end of the program, add:

```
dbmclose( %issecretword );
```

- Now, the file reading is done entirely automatically for you.
- But more efficiently: the DBM file is not a plain text file. In a large DBM file containing millions of **(key, value)** pairs, retrieving the value corresponding to a specific key is usually done using **only two disk accesses**!
- By using this highly efficient system, we get *persistent storage* for Perl programs, for free!
- This is an example of what I meant by leverage.

- Let's close by showing the final version **eg10**:

```
#
# eg10: secret words from a dbm file
#
dbmopen( my %issecretword, "secretwords", 0666 ) || die;

print "Please enter your name: ";
my $name = <STDIN>;
chomp $name;
$name = ucfirst(lc($name));
if( $name =~ /^Dun[ck]/ )
{
        print "\nwotcha Dunc mate!\n";
} else
{
        print "\nhello $name!\n";
        print "Please enter one of the secret words: ";
        while(1)
        {
                my $guess = <STDIN>;
                chomp $guess;
        last if $issecretword{$guess};
                print "Wrong - guess again: ";
        }
}

dbmclose( %issecretword );
```