

# Funda- mentals of Perl

© Bernd Klein

**Redistribution for  
non-commercial  
use without  
changes and with  
author's credit  
allow.**

The chief architect  
of Perl:  
Larry Wall

He designed,  
created (1987)  
and is maintaining  
Perl





**Perl 1.0**

December 18, 1987

**Perl 2.0** (1988)

better regular expression

**Perl 3.0** (1989)

support for binary data streams

Programming

**Perl**

de facto reference

**1991****Perl 4.0** (1993)

very shortlived

**Perl 5.0** (1994)

completely new interpreter

# History of Perl

**Perl 5.10.0**  
**(Dec 12, 2007)**

# Parable of the Pe(a)rl



*Again, the kingdom of heaven is like unto a merchant man, seeking goodly pearls:*

*Who, when he had found one pearl of great price, went and sold all that he had, and bought it.*

*(Gospel of Matthew, 13:45-46)*

# Perl is also an acronym

P  
ractical E  
xtraction and R  
eport L  
anguage

it should never ever be spelled in caps

it is considered a shibboleth  
(branding one as an outsider of the Perl community)



# Family

Perl is a dynamic programming language borrowing from many other programming languages, such as

C

Bourne and other Shell scripting languages

AWK

Sed

Lisp



# Perl and Lisp

## What Makes Lisp Different?

“Lisp has all the visual appeal of oatmeal with fingernail clippings mixed in.” (Larry Wall)

“Paradigms of Artificial Intelligence Programming” by Peter Norvig, includes a section titled “What Makes Lisp Different?” that describes seven features of Lisp. Perl shares six of these features; C shares none of them.

# The Seven Features of Lisp

- Built-in Support for Lists
- Automatic Storage Management
- Dynamic Typing
- First-Class Functions
- **Uniform Syntax**
- Interactive Environment
- Extensibility
- History

# Features

Perl is basically a procedural language similar to C with

- variables
- expressions
- assignment statements
- brace-delimited code blocks
- control structures
- subroutines

Other important features to simplify and facilitate many parsing, text handling, and data management tasks.:

- **lists** from Lisp,
- **associative arrays** (hashes) from AWK,
- **regular expressions** from sed.

# Features continued

In Perl 5 some features were added that support

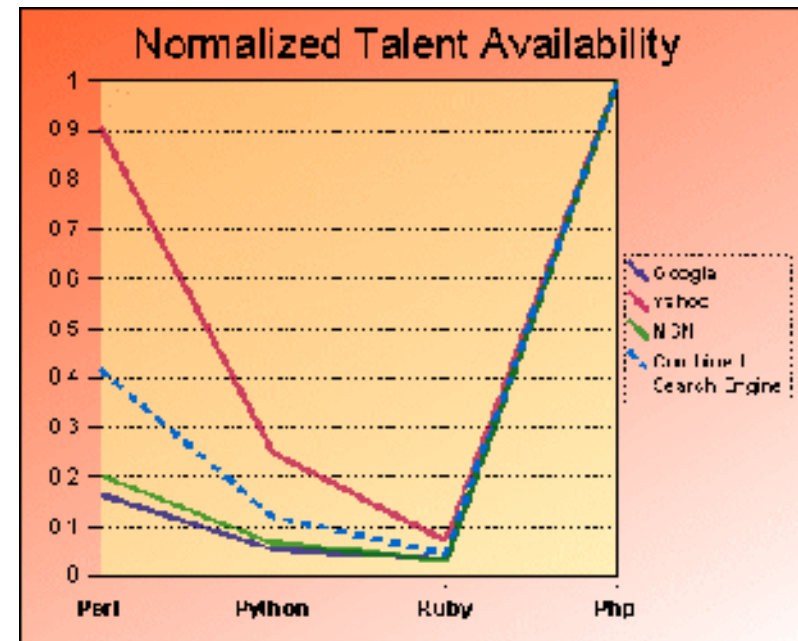
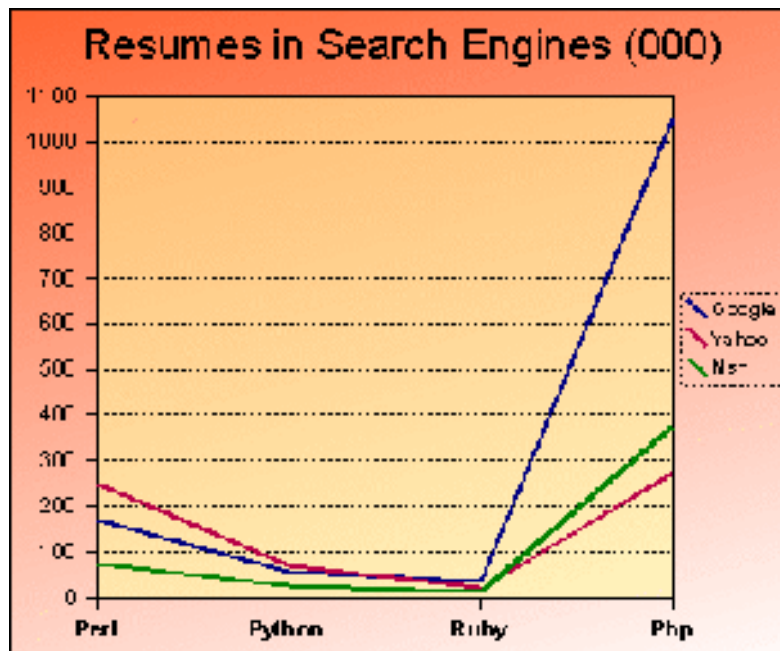
- complex data structures,
- first-class functions (i.e., closures as values),
- object-oriented programming model. These include references, packages, class-based method dispatch, and lexically scoped variables,
- compiler directives (for example, the strict pragma).
- ability to package code as reusable modules.



# Availability of Perl Programmers

Availability of experienced and talented developers is a key requirement of corporate adoption and retention of a programming language.

(intitle:resume OR inurl:resume) LanguageName -intitle:jobs -resumes -apply



Quelle: <http://www.odinjobs.com>, Nov. 2007  
Naveen Bala

# Implementation

Perl is an interpreted language.

The core interpreter is written in C

Plus a large collection of modules both written in C and Perl.

12 MB when packaged in a compressed tar file.

- compiled about 1 MB
- 150,000 lines of C code architectures.
- nearly 500 modules in the Perl-distribution, comprising 200,000 lines of Perl and an additional 350,000 lines of C code.

# Ways to run Perl

- Running the perl script included in the command line line by line via the -e switch:

```
perl -e 'print "Hello, world\n"'      #Unix
perl -e "print \"Hello, world\n\""    #Windows
```

- Running the perl command with the Perl program supplied via the standard input stream.

```
echo "print 'Hello, world'" | perl -
```

or (unless ignoreeof is set):

```
% perl
print "Hello, world\n";
^D
```

# Exercise

Test the following command lines in a shell:

```
perl -e 'print "Hello, world\n"'  
or  
echo "print 'Hello, world'" | perl -
```



# Hello World in Java

It can be difficult just to say “Hello World”:

```
class HelloWorldApp {  
    public static void main(String[] args) {  
        System.out.println("Hello World!"); //  
        Display the string.  
    }  
}
```

# Ways to run Perl, continued

- Issue the perl command, passing Perl the name of your script as the first parameter (after any switches):

```
perl testpgm
```

- On Unix systems: usage of a special #-Line.

# Hello, world!

shebang line



```
#!/usr/bin/perl
```

```
print "Hello, world!\\n";
```

used to separate  
statements

## Exercise

Use an editor to type in the „Hello Word“-Programm and save it as hello.pl

```
#!/usr/bin/perl  
print "Hello, world!\n";
```

Invoke the programm by typing in a shell:

```
perl hello.pl
```

execute the following command to run the script directly:

```
chmod a+x hello.pl  
hello.pl
```



# How to ask questions ...

**... and  
remember  
the  
results!**



# Input and output

We'll continue with a personalized version of „Hello World“, which greets the user instead of the whole world

We need a variable to hold a value in the following example.

We'll use `$name`, a scalar variable.

The `<STDIN>`-construct is a way to get a line from the terminal. (`<>` the diamond operator)

```
print "What is your name? ";  
$name = <STDIN>;
```

# Exercise

Save the previous script as `input1.pl`

```
print "What is your name? ";  
$name = <STDIN>;
```

Invoke the program!

# Improving the previous example

The program should output the Name together with an „hello“, which can be done with the following print statement:

```
print "Hello, $name!\n";
```

If we append this line to the previous code, we get the following output, if we use „Larry“ as input:

```
What is your name? Larry  
Hello, Bernd  
!
```

## „a tiny flaw“

The exclamation mark is in the following line and not behind the name.

The reason:

The value of `$name` has a terminating newline „`\n`“.

`chomp` is a special function, which takes a scalar variable as its sole argument and removes the trailing newline from the string value of the variable.



# The complete script

```
#!/usr/bin/perl -w
print "What is your name? ";
$name = <STDIN>;
chomp ($name);
print "Hello, $name!\n";
```

# Command-Line Options

The command-line options (or switches, flags) come first on the command line.

The next item is usually the name of the script, followed by any additional arguments (e.g. filenames) to be passed into the script.

# Some Perl Switches

- forces switch processing to terminate, even if the next argument starts with a minus.
- c the syntax of the script will be checked but the script will not be executed.
- d runs the script under the Perl debugger
- h prints a summary of command-line options
- v prints the version and patch level of your Perl executable
- V Summary of the most important Perl configuration values
- w warnings about unset variables, redefined subroutines etc.

# Perl compiler's C backend

```
perl -MO=C[,OPTIONS] foo.pl
```

This compiler backend takes Perl source and generates C source code corresponding to the internal structures that perl uses to run your program. When the generated C source is compiled and run, it cuts out the time which perl would have taken to load and parse your program into its internal semi-compiled form. That means that compiling with this backend will not help improve the runtime execution speed of your program but may improve the start-up time. Depending on the environment in which your program runs this may be either a help or a hindrance.

# Compiling Perl into C with `perlcc`

```
perlcc HelloWorld.pl -o Hello
```

This command generates an executable “Hello World” file.

It's possible to create the C source files with the `-S` option:

```
perlcc -S HelloWorld.pl
```



# Exercise

Create C source code and executables of some perl scripts, e.g. HelloWorld.pl !

# Variables

A variable always begins with the character that identifies its type:

\$ , @ , or %

Most of the variable names you create can begin with a letter or underscore, followed by any combination of letters, digits, or underscores, up to 255 characters in length.

Upper- and lowercase letters are distinct.

# undef value

Variables have the `undef` value before they are first assigned or when they become "empty." For scalar variables, `undef` evaluates to zero when used as a number, and a zero-length, empty string (`""`) when used as a string.

# Variable assignment

The assignment operator (=) is used for variable assignment with the appropriate data.

It takes a variable on the left side and gives it the value of the expression on the right:

```
$count = 0;
```

```
$res = $b * ($a + 2.4534);
```

# Assignments used as a value

An assignment has a value as well, i.e. the value is the value assigned.

Example:

```
$a = 1 + ($b = 7);
```

The value 7 is assigned to \$b, then the value 7 as the value of the assignment is added to 1 and 8 will be assigned to \$a.

# Data types

- A **scalar** is a single value; it may be a number, a string or a reference
- An **array** is an ordered collection of scalars
- A **hash**, or associative array, is a map from strings to scalars; the strings are called keys and the scalars are called values.
- A **file handle** is a map to a file, device, or pipe which is open for reading, writing, or both.
- A **subroutine** is a piece of code that may be passed arguments, be executed, and return data



# Sigils to identify data type

`$foo` # a scalar

`@foo` # an array

`%foo` # a hash

`FOO` # a file handle or constant

`&foo` # a subroutine.  
# The & is optional)

File handles and constants need not be uppercase, but it is a common convention owing to the fact that there is no sigil to denote them.

# Scalar Data

A scalar is essentially a simple variable, i.e. the simplest kind that Perl manipulates.

A scalar can be

- a number (like 7 or 8.2620) or
- a string of characters (like „hello“ or „data types and variables“).

Though you might think of numbers and strings as very different things, Perl uses them nearly interchangeably.

## Scalar Variables, cont.

A variable is a name for a container that holds a value.

The name of the variable is constant throughout the program, but the value contained in that variable can change over and over again throughout the execution of the program.

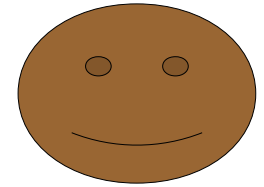
A scalar variable holds a single scalar value, which can be a number, a string, or a reference. Scalar variable names begin with a dollar sign followed by a letter, and then possibly more letters, or digits, or underscores.

Perl is case sensitive, so upper- and lowercase letters are distinct.

# Exercise

What is the value of \$a after the assignment?

`$a = 024 + 026;`



42

# Numbers

```
42          # an integer
-42         # a negative integer
3.1415      # a floating point
4.35E32     # scientific notation 4.35 times 10
to the 32th
0xff        # hexadecimal literal
012         # octal literal

23_234_767  # underline for legibility
```

## important note:

The underscore only works within literal numbers specified in the program, but not in strings functioning as numbers or in data read from somewhere else.

Similarly, the automatic conversion of a string to a number does not recognize the „0“ and „0x“-prefixes.

# Same Internal Format

Internally, Perl computes only with **double-precision floating-point values**.

This means that Perl doesn't use integer values internally.

Consequently, Perl doesn't supply special integer operations.



# Float Literals

A literal is the representation of a value in the text (source code) of the Perl program, in other words a *constant*.

Perl accepts the complete set of floating-point literals to C programmers.

## Float Literals

1.75

7.25e45

-6.5e24

-12e-24

-1.2E-23

## Integer Literals

1

2

-1

42

1\_453\_987\_112

# Strings

A string consists of a sequence of characters.

Each character is an 8-bit value from the 256 character set  
(no special meaning of the NUL character)

The shortest possible string contains no characters.

The longest string is determined by the available memory.

# Double-Quoted Strings

The double-quoted string is similar to a C string.

It's a sequence of characters enclosed in double quotes.

The backslash can be used to specify certain control characters.

## Examples of Double-Quoted Strings

`"hello world\n"`     # string with a newline

`"umlaut \334"`     # umlaut ü (oktal)

`"km\t1089"`     # a tab in a string

# Single Quoted-Strings

A single-quoted string is a sequence of characters enclosed in single quotes.

The single quotes are marking the beginning and the ending of the string, but are not part of the string itself.

Any character between the single quote marks is legal inside a string.

## **Exceptions from the rule:**

Backslashes have no special meaning, except if a backslash is followed by another backslash:

To get a single quote into a single-quoted string, precede it by a backslash.

# Examples of Single-Quoted Strings

`'Perl'`           # four characters: P, e, r, l

`'don\'t'`       # five characters:  
                  d, o, n, single-quote, t

`''`               # the empty/null string

`'home\\eve'`   # includes one backslash

`'hello\n'`      # not a newline!

# String Conversions

Perl converts strings into numbers and vice versa depending on the context in which they are used.

What is printed in the following script?

```
$fruit = "3 peaches";  
$vegetable = "2 cabbages";  
print $fruit + $vegetable;
```

The output is '5', the non number information is discarded.

The string concatenation operator is „.“ and not „+“



# Exercises

- Write a program that prompts for and reads two numbers (on separate lines of input) and prints the product of the two numbers.
- Write a program using string conversions.
- Experiment with single-quoted String and double-quoted strings in a program, i.e. assign string to variables and print them.

# Scalar Operators

An operator expects either numeric or string operands or a combination of both.

If a string operand is provided where a numeric value is expected, or vice versa, Perl automatically converts the operand.

# Numeric Operators

The ordinary operators for

- addition:  $+$
- subtraction:  $-$
- multiplication:  $*$
- division:  $/$

other operators:

exponentiation:  $**$

modulus:  $\%$

logical comparison:  $<$ ,  $<=$ ,  $==$ ,  $>=$ ,  $>$ ,  $!=$



# Precision

What makes  $7.1 - 1.3$  ?

## 5.8

Are you sure?

Isn't it rather ...

**5.799999999999999999999998223643161**

```
perl -e 'printf("%.25f\n", 7.1 - 1.3)';
```

# Operators for Strings

Concatenation: “.” operator

Concatenating string with the “.” operator does not alter either of the involved strings.

Examples:

```
"homer" . " " . "simpson"
```

```
"hello world" . "\n"
```

# String Comparison

Equal	<code>eq</code>
Not equal	<code>ne</code>
Less than	<code>lt</code>
Greater than	<code>gt</code>
Less than or equal to	<code>le</code>
Greater than or equal to	<code>ge</code>

# String Repetition Operator

The single lowercase letter `x` denotes the string repetition operator. It takes its left operand (a string), and concatenates as many copies of this string together as indicated by its right operand, which is treated as a numeric value.

`"Homer" x 3`  `"HomerHomerHomer"`

`"Marge" x (1 + 1)`  `"MargeMarge"`

`(3 + 2) x (2.3 * 3)`  `"555555"`



# Exercise

What is the value of the following comparisons?

8 < 30            true

8 lt 30            false

# Lists and Arrows

A list is ordered scalar data. An array is a variable that holds such a list.

Each element of an array is a separate scalar variable with its own scalar value.

The elements are ordered by indices 0,1,2, ...

Arrays can have any number of elements.

The smallest array has no elements, while the largest array can fill all of available memory.

# Arrays

The values of a list can be

- numbers,
- strings, or even
- another array.

Array variable names always begin with a @.

(Mnemonic: The @ sign starts an array variable because “at” and “array” start with the same letter)

## Example of an Array

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

@A = ( " or ", " not " );
@prefix = ( 2, "B " );
@suffix = @prefix;
@william = (@prefix, @A);
@william = (@william, @suffix);


print "@william\n";
```


# List Constructor Operator

A list constructor operator is construed by two scalar values separated by two consecutive periods.

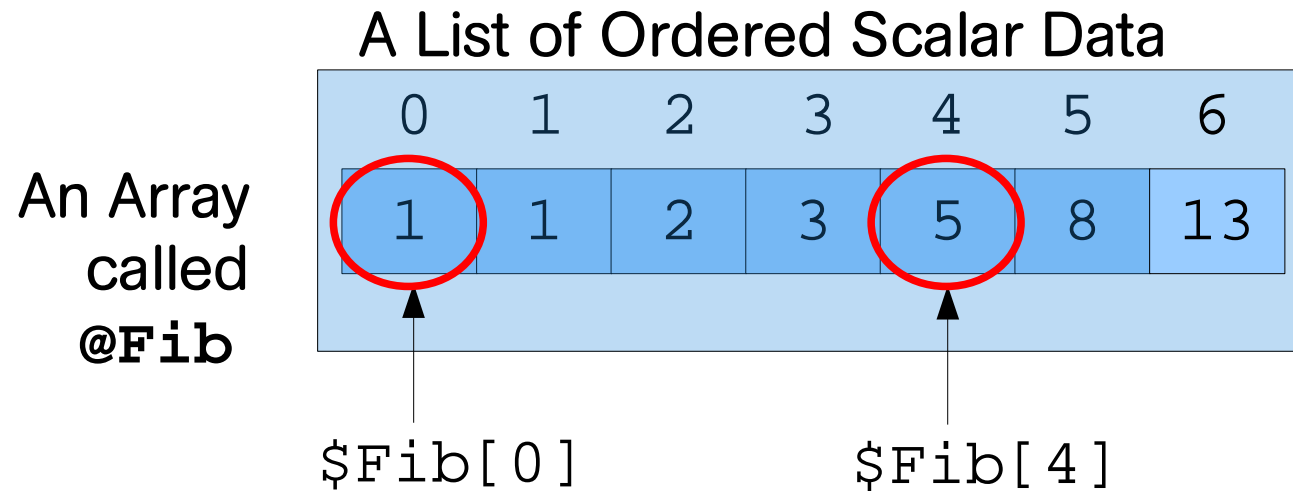
A list of values is created by starting at the left scalar value up through the right scalar value, incremented by one each time.

( 1 .. 5 )            ( 1 , 2 , 3 , 4 , 5 )

( 2 . 3 .. 5 . 3 )            ( 2 . 3 , 3 . 3 , 4 . 3 , 5 . 3 )

( 2 . 3 .. 6 . 2 )            ( 2 . 3 , 3 . 3 , 4 . 3 , 5 . 3 )

# Array Element Access



The `@` of the array name becomes a `$` on the element reference!

# Exercise

**What's printed in the following script?**

```
@Fib = (1, 1, 2, 3, 5, 8, 13, 21)
```

```
$i = 5;  
$a = $Fib[4];  
$b = $Fib[$i]++;  
$c = $Fib[$i++];  
$i = 5;  
$d = $Fib[++$i];  
print "$a, $b, $c, $d\n";  
print "@Fib\n";
```



# Syntactic Sugar for List Literals

instead of defining an array like this:

```
@languages = ( "english", "french", "german" );
```

one can use the “quote word” function:

```
@languages = qw(english french german);
```

# Slice

Accessing a list of elements from the same array is called a slice:

```
@Fib = (1, 1, 2, 3, 5, 8, 13, 21);  
($Fib[1], $Fib[2]) = ($Fib[2], $Fib[1]);  
print "@Fib\n";
```

Syntactic sugar for slices:

```
@Fib[0,1] = @Fib[1,0] # swap the first two el.  
@Fib[0,1];           # same as  
                    # ($Fib[0], $Fib[1])  
@Fib[0,1,2] = @Fib[1,1,1] # first three like 2nd
```

# Arrays with no Boundaries

Assigning a value to an index beyond the existing ones automatically extends the array (giving a value of undef to all intermediate values, if there are any).

```
@b = ( "to" , "be" );  
$b[2] = "or" ;  
$b[5] = "be" ;  
  
print "@b\n" ;
```

`$b[3]` and `$b[4]` will have the value "undef".

Arrays  
can be  
seen or  
used as  
stacks!



## push and pop

An array can be used like a stack of information. New elements are removed from and added to the right-hand side of the list. `pop` and `push` are introduced into Perl to facilitate this task:

```
push(@list, some_value);
```

is equivalent to

```
@list = (@list, some_value);
```

`pop` removes the last element of a list (array):

```
$last_element = pop(@list);
```

# Shift and Unshift

The push and pop functions operate from the "right" side of a list, i.e. the one with the highest subscripts.

The unshift and shift functions perform the corresponding actions on the "left" side of a list, i.e. the one with the lowest subscripts.

# Shift and Unshift, Example

```
@colors = ("red", "green", "blue");  
$y = "yellow";  
unshift(@colors, $y, "white", "black");  
print "colors: @colors\n";  
$most_left = shift(@colors);  
print "most_left: $most_left\n";  
print "colors: @colors\n";
```

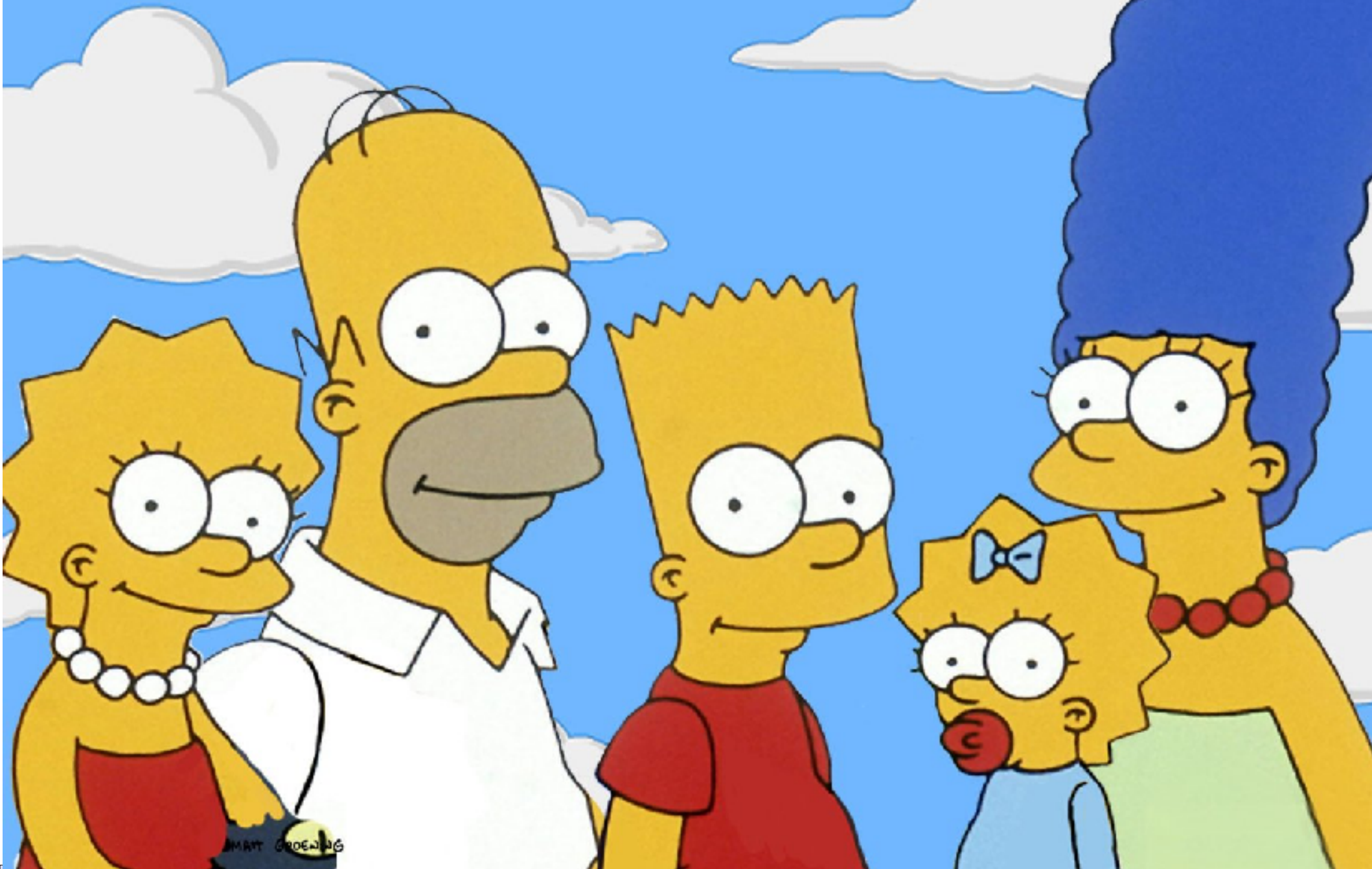
```
# remove the most left value and append it to  
# the right side:  
push(@colors, shift(@colors));  
print "colors: @colors\n";
```



# The reverse Function

The reverse function returns a list in which the order of the elements of its argument are reversed..

**@simpsons = ("Lisa","Homer","Bart","Maggie","Marge");**



```
@simpsons = ("Lisa","Homer","Bart","Maggie","Marge");  
@simpsons = reverse(@simpsons );
```





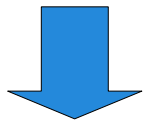
# sort Function

The sort function sorts its arguments as if they were single strings in ascending ASCII order.

It returns the sorted list without altering the original list.

```
@simpsons = sort("Homer", "Marge", "Bart", "Lisa", "Maggie");  
print "in alphabetical order: @simpsons\n";
```

```
@Fib = (1, 1, 2, 3, 5, 8, 13, 21);  
@sorted = sort(@Fib);  
print "@sorted\n";
```



1 1 13 2 21 3 5 8

# The chomp Function

The `chomp` function works both on array variables and on scalar variables.

It takes off the end character of a specified string ONLY if that character is a RETURN (Enter).

The return character is often created from input information or by the coding itself.

It doesn't affect any other characters.

```
@simpsons=( "Lisa\n" , "Homer\n" , "Bart" , "Maggie\n" , "Marge" ) ;  
@simpsons  = chomp(@simpsons ) ;
```

`@simpsons` is now:

```
( "Lisa" , "Homer" , "Bart" , "Maggie" , "Marge" )
```

# The chop Function

The chop function is very similar to Chomp, but it “chops off” the ending character of a string no matter what it is.

## <STDIN> as an Array

<STDIN> returns the next line of input in a **scalar context**.

However, in a **list context**, it returns all remaining lines up to end of file.

Each line is returned as a separate element of the list.

```
@a = <STDIN>;  
print "@a\n";
```

## Exercise

Write a script that reads a list of strings on separate lines and prints out the list in reverse order.



# Hashes

Hashes are better known as **associative arrays**.

When defining a whole hash, we use the same representation that we use for arrays – but we need two items for every element in the associative array.

Values to individual elements of an hash are assigned by using curly braces ({} ) around the index key.

Example:

```
%birthdays = ( "Mike", "Apr 29", "Jane", "Feb 1",  
"Freddy", "Dec 7" );  
print "Freddys birthday is:" . $birthdays{"Freddy"}  
. " \n";
```

# Exercise

Write a program that will ask the user for a given name and report the corresponding family name.


# Control Structures

- Statement blocks
- conditional statements:
  - if/unless statement
- Loops:
  - while/until statement
  - for statement
  - foreach statement

# Statement Blocks

A statement block is a sequence of statements, surrounded by matching curly braces:

```
{  
    first_statement;  
    second_statement;  
    third_statement;  
    ...  
    last_statement;  
}
```

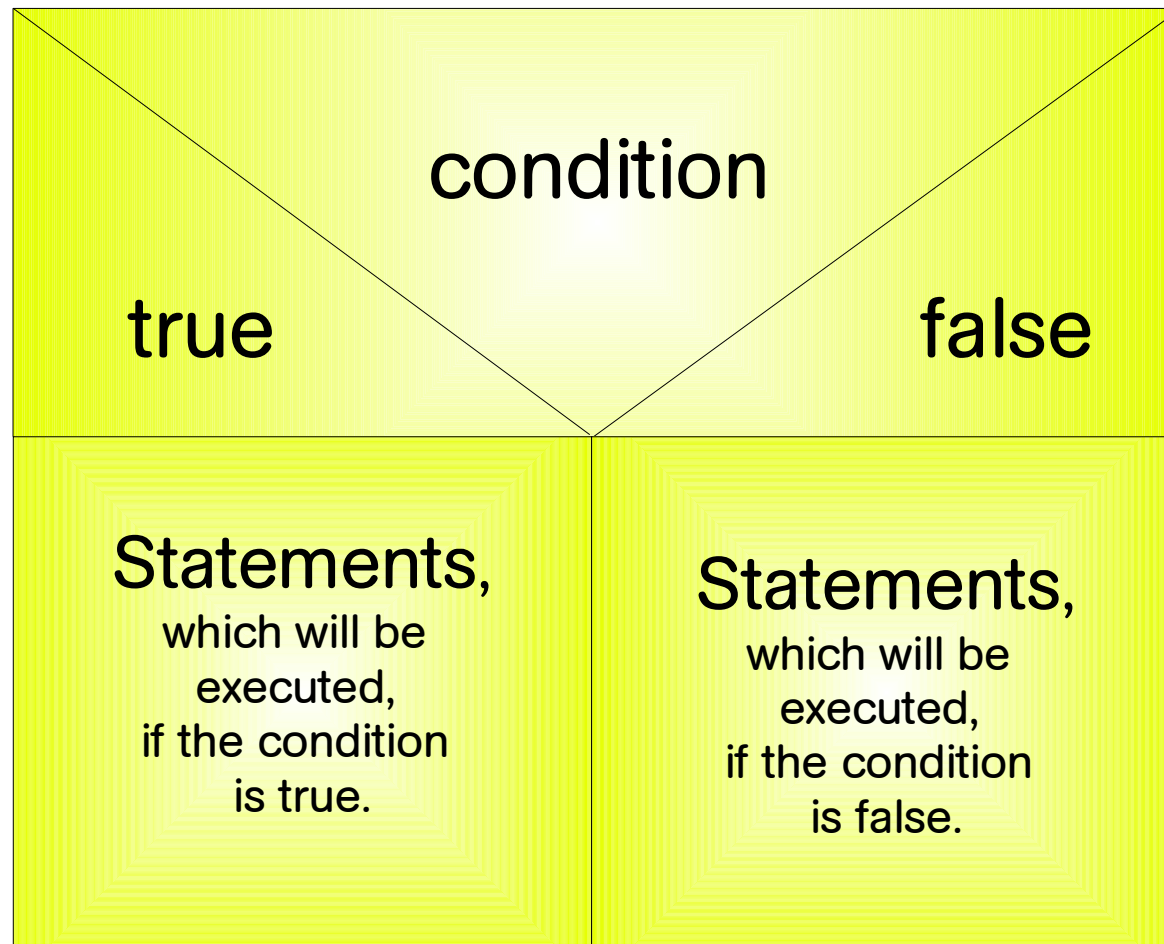


Each statement will be executed in sequence, from the first to the last.



# Conditional Statements





# The if/unless Statement

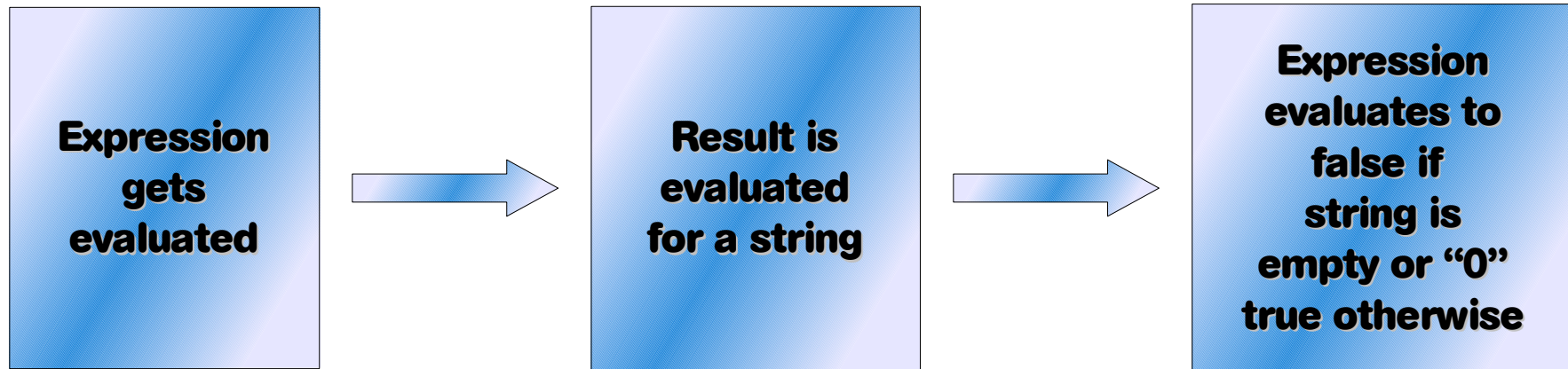
The if construct takes a control expression and a block. optionally, it may have an else followed by a block as well:

```
if (some_expression) {  
    true_statement_1;  
    true_statement_2;  
    true_statement_3;  
} else {  
    false_statement_1;  
    false_statement_2;  
    false_statement_3;  
}
```

**(Attention:** Curly braces are required.  
This eliminates the need for a “confusing dangling else” rule.)



# The Control Expression



0	false, as 0 converts to "0"
3-3	computes to 0, then converts to "0", so false
" "	empty string, so false
"1"	neither "" nor "0", so true
1	converts to "1", so true
"00"	neither "" nor "0", so true
"0.000"	also true for the same reason
undef	evaluates to "", so false



## Example of an `if` statement

```
print "How old are you? ";
$age = <STDIN>;
chomp($age);
if ($age < 18) {
    print "Sorry, not old enough to vote!\n";
} else {
    print "Old enough! So go vote!\n";
    $possible_voter++;
}
```



# Omitting the `else` part

```
print "How old are you? ";  
$age = <STDIN>;  
chomp($age);  
if ($age < 18) {  
    print "Sorry, not old enough to vote!\n";  
}
```



# unless

If you just want the else part and no then part, you can use the unless statement:



```
print "How old are you? ";
$age = <STDIN>;
chomp($age);
unless ($age < 18) {
    print "Old enough! So go vote!\n";
    $possible_voter++;
}
```

# elsif for more than two choices

```
if (expression_1) {  
    one_true_statement_1;  
    one_true_statement_2;  
    one_true_statement_3;  
}  
elsif (expression_2) {  
    two_true_statement_1;  
    two_true_statement_2;  
    two_true_statement_3;  
}  
elsif (expression_3) {  
    three_true_statement_1;  
    three_true_statement_2;  
    three_true_statement_3;  
}  
else {  
    all_false_statement_1;  
    all_false_statement_2;  
    all_false_statement_3;  
}
```

Every expression (expression\_1, expression\_2 and expression\_3) is computed in turn. If an expression is true, the corresponding branch is executed, and all remaining control expressions and corresponding statement blocks are skipped. If no expression evaluates to true, the else branch is executed, if there is one.

# Switch Statement

Perl has no switch statements!

Solution: Hashes!

Use subroutine references in a hash to define what to do for each case.

```
$action_to_take = (  
    1 => \&process_direct_deposits,  
    2 => \&query_account_status,  
    3 => \&do_exit,  
);
```

Call with (instead of if):

```
$action_to_take{$menu_item}->();
```

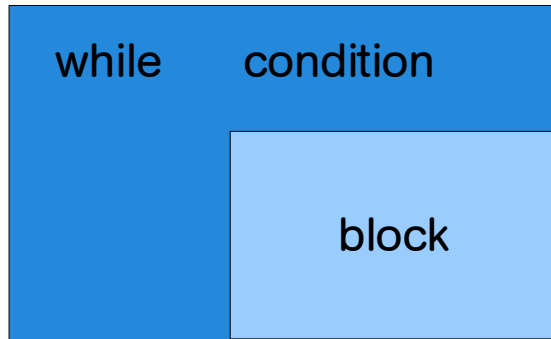
# Loop Statements

Most programming languages need loops to repeat the execution of a statement block while some condition is true.

There are three types of loops:

- while loops,
- until loops, and
- for loops.

# while Statement



```
while (expression) {  
    statement_1;  
    statement_2;  
    statement_3;  
}
```

Perl evaluates the control expression `expression`. If its value is true, the body of the while statement (the block) is evaluated once.

This is repeated until the control expression becomes false, at which point Perl goes on to the next statement after the while loop.

# until Statements

If the word `while` is replaced by the word `until`, the test is reversed (negated); that is, it executes the block as long as `EXPR` remains false.



**The condition is also tested before the first iteration.**

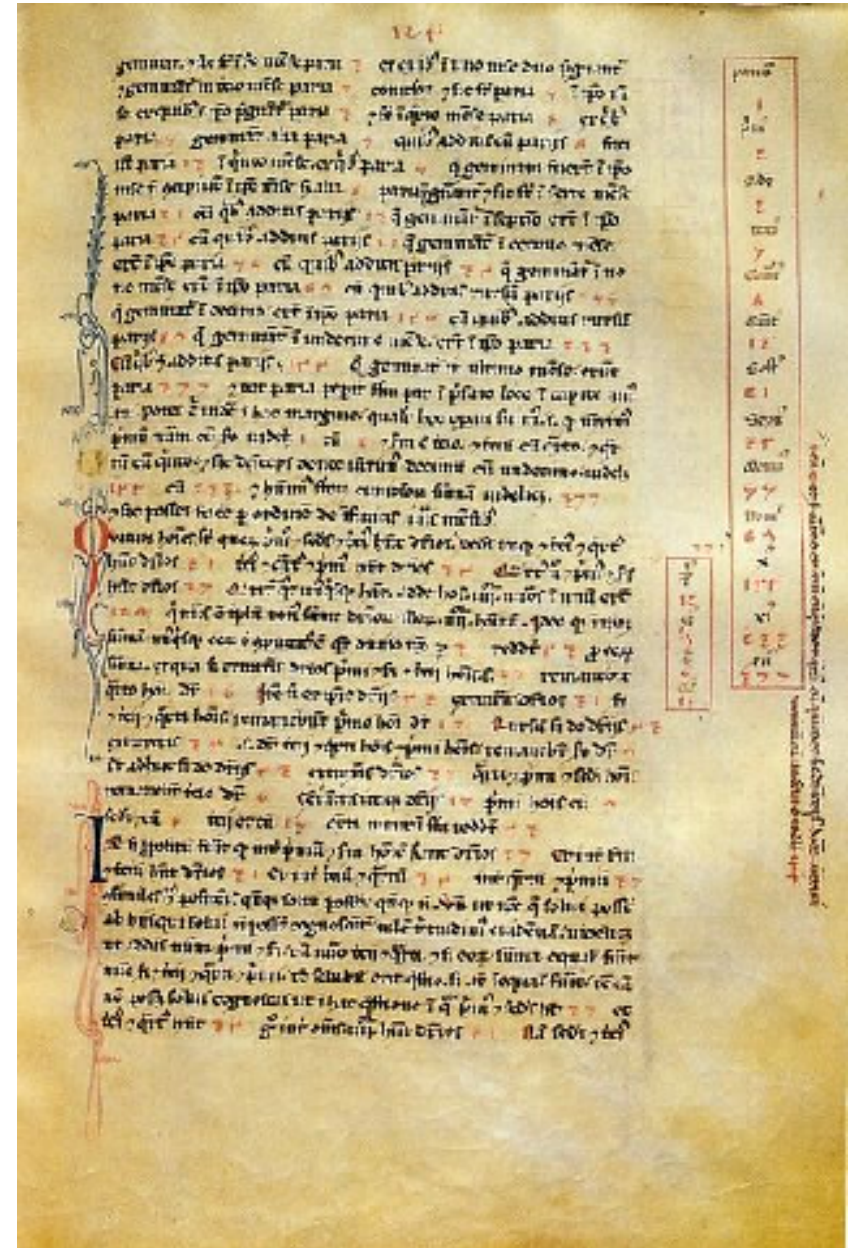


# Leonardi Fibonacci

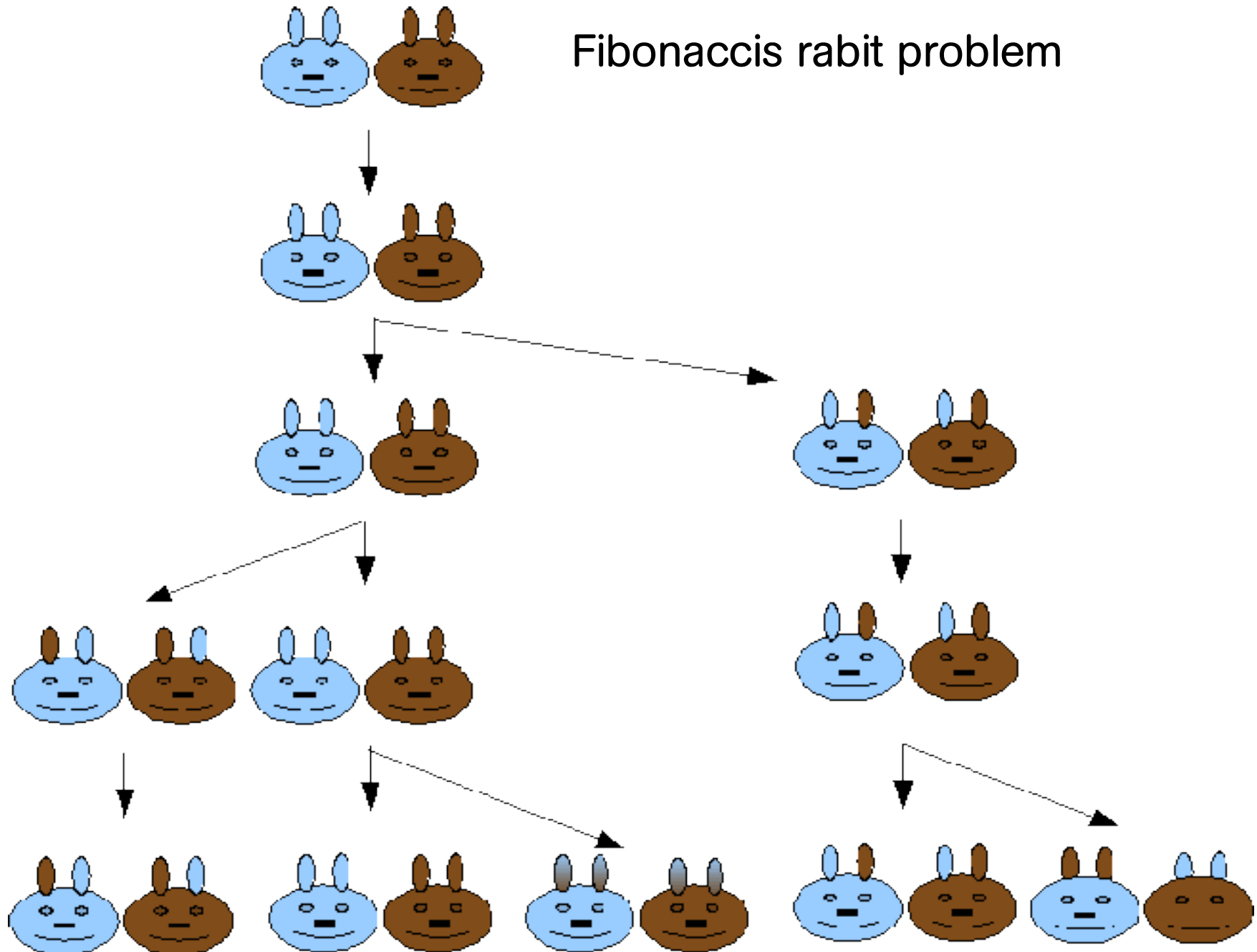
Leonardo da Pisa,  
known as Fibonacci  
figlio de Bonacio  
(1180 - 1241)

one of the most important  
mathematicians of the  
Middle Ages

most important work:  
Liber abbaci



# Fibonacci's rabbit problem



# Exercise

**Definition:** A member of the sequence of numbers such that each number is the sum of the preceding two. The first seven numbers are 1, 1, 2, 3, 5, 8, and 13.

**Formal Definition:**

The nth Fibonacci number is  
 $F(n) = F(n-1) + F(n-2)$ ,  
where  $F(1)=1$  and  $F(2)=1$

**Task:**

Write a script to calculate  $F(n)$ !

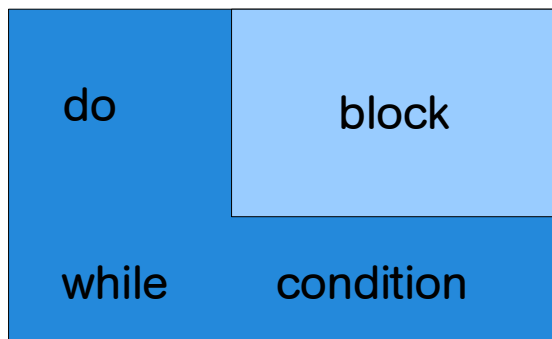
```
#!/usr/bin/perl -w
print "Number? ";
$n = <STDIN>;
chomp($n);
if ($n >= 1) {
    $i = 1;
    $fib_n_1 = 0;
    $fib_n = 1;
    while ( $i < $n ) {
        $help = $fib_n + $fib_n_1;
        $fib_n_1 = $fib_n;
        $fib_n = $help;
        print "schnitt: " . $fib_n / $fib_n_1 . "\n";
        $i++;
    }
    print "Fibonacci number $n: $fib_n\n";
} else {
    print "Please start with a number >= 1\n";
}
```

## Solution

# do {} while/until Statements

```
do {  
    statement_1;  
    statement_2;  
    statement_3;  
} while expression;
```

This construct is similar to the `while/until` statement, previously discussed, but the block will be - regardless of the starting value of the expression - at least once executed.



# from while to for

Somebody working solely with `while` loops will often encounter typical structures like this:

```
# initialization of a loop variable
# affecting the test_exp:
$i = 0;
while (test_exp) {
    # a sequence of statements:
    statement_1;
    statement_2;
    ...
    # incrementing or decrementing the
    # loop variable
    $i++;
}
```



# for statement

```
for ( <INITS>; <TEST_EXPR>; <RE_INITS> ) {  
    # body with statements  
}
```

initialization

condition

re-initialization

<INITS> consists of a comma separated list of loop variable initializations.

<TEST\_EXPR> is a condition which is affected by the variables defined in <INITS>

<RE\_INITS> consists of a comma separated list of assignments to the loop variables.

# Example

The following script prints the square numbers from 1 to 100.

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

for ($i=1; $i < 100; $i++) {
    print "$i: " . $i**2 . "\n";
}
```



# infinite loops with `while` and `for`

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

```
while (1) {  
  
}
```

# foreach Loops

The foreach loop iterates over a list of values by setting a control variable to each successive element of the list:

```
foreach $VAR (LIST) {  
    . . .  
}
```

It's not necessary to use the `foreach` keyword as `foreach` is just a synonym for the `for` keyword.

# Example

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

@l = (3, 4, 6, 8);
foreach $i (@l) {
    print $i**2 . "\n";
}
```

A “slight” modification with grave consequences:

```
@l = (3, 4, 6, 8);
foreach $i (@l) {
    $i = $i**2 . "\n";
    print $i;
}
```

@l has changed to (9, 16, 36, 64)

# Labelling Loops

A label can be put on a loop to name it.

This label identifies the loop for the loop-control operators

- next
- last
- redo

If the loop operators are used with a label, they use the loop with this label otherwise the operator refers to the innermost enclosing loop.

## next

```
next [LABEL]
```

The next command is like the continue statement in C; it starts the next iteration of the loop:

```
line: while (<STDIN>) {  
    next line if /^#/;      # discard comments  
    ...  
}
```

If there exists a continue block, it will get executed even on discarded lines. If the LABEL is omitted, the command refers to the innermost enclosing loop.

## Example with next

```
$n = <STDIN>; # the final number
print "$n\n";
$count = 0;
while ($count <= $n) {
    # set a conditional statement to interrupt
    if (( ($count % 4) == 0) && ($count != 0)) {
        print "$count is divisible by 4!\n";
        $count ++;
        next;
    }
    # go on as usual
    print $count."\n";
    $count ++;
}
continue {
    print $count."\n";
    $count++;
};
print "That's all! Task finished!";
```

# last

```
last [LABEL]
```

The `last` command is like the `break` statement in C (as used in loops); it immediately exits the loop in question.

If the `LABEL` is omitted, the command refers to the innermost enclosing loop.

The `continue` block, if any, is not executed:

```
line: while (<STDIN>) {  
    last line if /^$/;    # exit when done  
                        # with header  
    ...  
}
```

## Example with next and last

```
my $count = 0;
LINE: while (<STDIN>) {
    next LINE if /^#/;    # skip comment lines
    next LINE if /^$/;    # skip blank lines
    print;    # $_
    last LINE if /eee/;
} continue {
    $count++;
}
print "$count lines have been input!\n";
```



# redo

```
redo [ LABEL ]
```

The redo command restarts the loop block without evaluating the conditional again. The continue block, if any, is not executed. If the LABEL is omitted, the command refers to the innermost enclosing loop.

# Example

```
# bad luck, an infinite loop:
$counter = 1;
while ($counter < 20) {
    redo if ($counter == 13);
    print "counter: $counter\n";
}
continue {
    $counter++;
};
```

# Bare Blocks

It's obvious, that a block, regardless if labeled or not, is semantically equivalent to a loop which executes just once.

## Exceptions:

- Not true for the blocks in `eval {}`, `sub {}` and `do {}`.
- loop controls don't work with `if` and `unless` either.

## Trick for `do {}`

By embedding a bare block inside the curly braces of a `do`-statements, we can use `next` and `redo` controls:

```
do {{
    next if .....;
    # some statements following
}} until .....;
```

A way to use `last`:

```
{
    do {
        last if ...;
        # some statements following
    } while ...;
}
```

## do {} Trick, continued

To include both next and last in the do {} we need labels.

```
DO_LAST: {  
    do {  
        DO_NEXT: {  
            next DO_NEXT if ...;  
            last DO_LAST if ...;  
            # some statements following  
        }  
    } while ...;  
}
```

# Case and Switch in Perl

**Perl has no case or switch statement unlike many other programming languages.**

But it's very easy to simulate the behaviour with bare blocks.

```
SWITCH: {  
    if (...) { $abc = 1; last SWITCH; }  
    if (/.../) { $def = 1; last SWITCH; }  
    if (/.../) { $xyz = 1; last SWITCH; }  
    $nothing = 1;  
}
```

# Case and Switch in Perl, cont.

an alternative:

```
SWITCH: {  
    /^abc/      && do { $abc = 1; last SWITCH; };  
    /^def/      && do { $def = 1; last SWITCH; };  
    /^xyz/      && do { $xyz = 1; last SWITCH; };  
    $nothing = 1;  
}
```

# Subroutines

A subroutine, also called sub or a “user function” is defined in a Perl program by using the following construct:

```
sub subname {  
    statement_1;  
    statement_2;  
    statement_3;  
}
```

The name of the subroutine (subname) which is any name like the names for scalar variables, arrays, and hashes.

The function names come from a different namespace, so a scalar variable `$foo`, an array `@foo`, a hash `%foo`, and now a `foo` would define different objects.



# Subroutine Example

```
sub hello_you_know_who {  
    $number_of_times_called++;  
    print "Hello World!\n";  
}
```

This subroutine is called like this:

```
hello_you_know_who( );
```

# Return Values

The value of the subroutine invocation is called the return value.

The return value of a subroutine is the value of the return statement or of the last expression evaluated in the subroutine.

```
sub areaOfCircle {  
    $radius = $_[0];  
    return(3.1415 * ($radius ** 2));  
}
```

# Positioning Subroutine definitions

Subroutine definitions can be placed anywhere in a script.

It's recommended to place the subroutines in one “area”, i.e. the beginning (C-Style) or at the end (Perl-Style).

Subroutine definitions are always global, i.e. there are no local subroutines.

In case there are two subroutines with the same name, the later one overwrites the earlier one.

By default, any variable used within the body of a subroutine refers to a global variable, but there are exceptions.

# Arguments

In Perl, the subroutine call is followed by a an (possibly empty) list within parentheses.

The values of this list are automatically assigned to a special variable named `@_` for the duration of the subroutine.

```
sub ListSum {  
    $sum = 0;           # initialize the sum  
    foreach $x (@_) {  
        $sum += $x;     # add element  
    }  
    return $sum;        # sum of all elements  
}  
  
print ListSum(1,2,3,4,5,6);
```

# Global and local Variables



# Subroutines: Local Variables

The `@_` variable is local to a subroutine,  
as well as `$_[0]`, `$_[1]`, `$_[2]`,

Other local variables can be declared with

```
local(<comma separated list of variable  
names>);
```

The `my` operator takes a list of variable names (instantiations) and creates local versions of them.

# Invoking a User Function

```
sub max {  
    if ($_[0] > $_[1]) {  
        $_[0];  
    } else {  
        $_[1];  
    }  
}  
  
print("enter first number: ");  
$number1 = <STDIN>;  
chomp($number1);  
print("enter second number: ");  
$number2 = <STDIN>;  
chomp($number2);  
  
$max = max($number1, $number2);  
print "max: $max\n";
```

# Difference between `my` and `local`

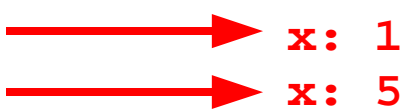
```
sub PrintX {  
    print "x: $x\n";  
}
```

```
sub UsingLocal {  
    local($x) = 5;  
    PrintX();  
}
```

```
sub UsingMy {  
    my($x) = 5;  
    PrintX();  
}
```

Variables declared with `local` are visible in the function where they are declared and in every function called by this function. Whereas values declared with `my` are only visible in the function in which they are declared.

```
$x = 1;  
PrintX();  
UsingLocal();
```



**x: 1**  
**x: 5**

```
$x = 1;  
PrintX();  
UsingMy();
```



**x: 1**  
**x: 1**



## Another difference

Whereas `my` can only be used to declare simple scalar, array, or hash variables with alphanumeric names, `local` doesn't have these restrictions.

Perl's built-in variables, such as `$_`, `$1`, and `@ARGV`, cannot be declared with `my`, but work fine with `local`.

# Command-line Arguments

With Perl, command-line arguments are stored in an array called `@ARGV`, i.e. `$ARGV[0]` contains the first argument, `$ARGV[1]` contains the 2nd argument, and so on.

```
#!/usr/bin/perl
```

```
$numArgs = $#ARGV + 1;  
print "$numArgs command-line arguments.\n";  
print "The arguments are:\n";  
foreach $argnum (0 .. $#ARGV) {  
    print "$ARGV[$argnum]\n";  
}
```

# Exercise: Ackermann function

**Write a program using a recursive function to calculate the Ackermann function:**

$$A(m, n) = \begin{cases} n + 1 & \text{if } m = 0 \\ A(m-1, 1) & \text{if } m > 0 \text{ and } n = 0 \\ A(m-1, A(m, n-1)) & \text{if } m > 0 \text{ and } n > 0 \end{cases}$$

# Ackermann Function in Perl

```
sub A {  
    local($x,$y,$x_1, $y_1);  
    $x = $_[0];  
    $y = $_[1];  
    if ($x == 0) {  
        $y+1;  
    } elsif ($y == 0) {  
        $x_1 = $x - 1;  
        A($x_1, 1);  
    } else {  
        $x_1 = $x - 1;  
        $y_1 = $y - 1;  
        A($x_1, A($x,$y_1));  
    }  
}
```

# Ackermann Function: Results

$n \setminus m$	0	1	2	3	4	$m$
0	1	2	3	4	5	$m + 1$
1	2	3	4	5	6	$m + 2$
2	3	5	7	9	11	$2m + 3$
3	5	13	29	61	125	$8 \cdot 2^m - 3$
4	13	65533	$2^{65536} - 3 \approx 2 \cdot 10^{19728}$	$a(3, 2^{65536} - 3)$	$a(3, a(4, 3))$	$2^{2 \cdots 2} - 3$ ( $m + 3$ Terme)
5	65533	$a(4, 65533)$	$a(4, a(5, 1))$	$a(4, a(5, 2))$	$a(4, a(5, 3))$	
6	$a(5, 1)$	$a(5, a(5, 1))$	$a(5, a(6, 1))$	$a(5, a(6, 2))$	$a(5, a(6, 3))$	

# Some Further Insides

addition:

$$a + b = a + \underbrace{1 + 1 + \dots + 1}_b$$

multiplication

$$a \times b = \underbrace{a + a + \dots + a}_b$$

exponentiation

$$a^b = \underbrace{a \times a \times \dots \times a}_b$$

tetration (hyper-4)

$${}^b a = \underbrace{a^{a^{\dots^a}}}_b$$

# Filehandles and File Tests

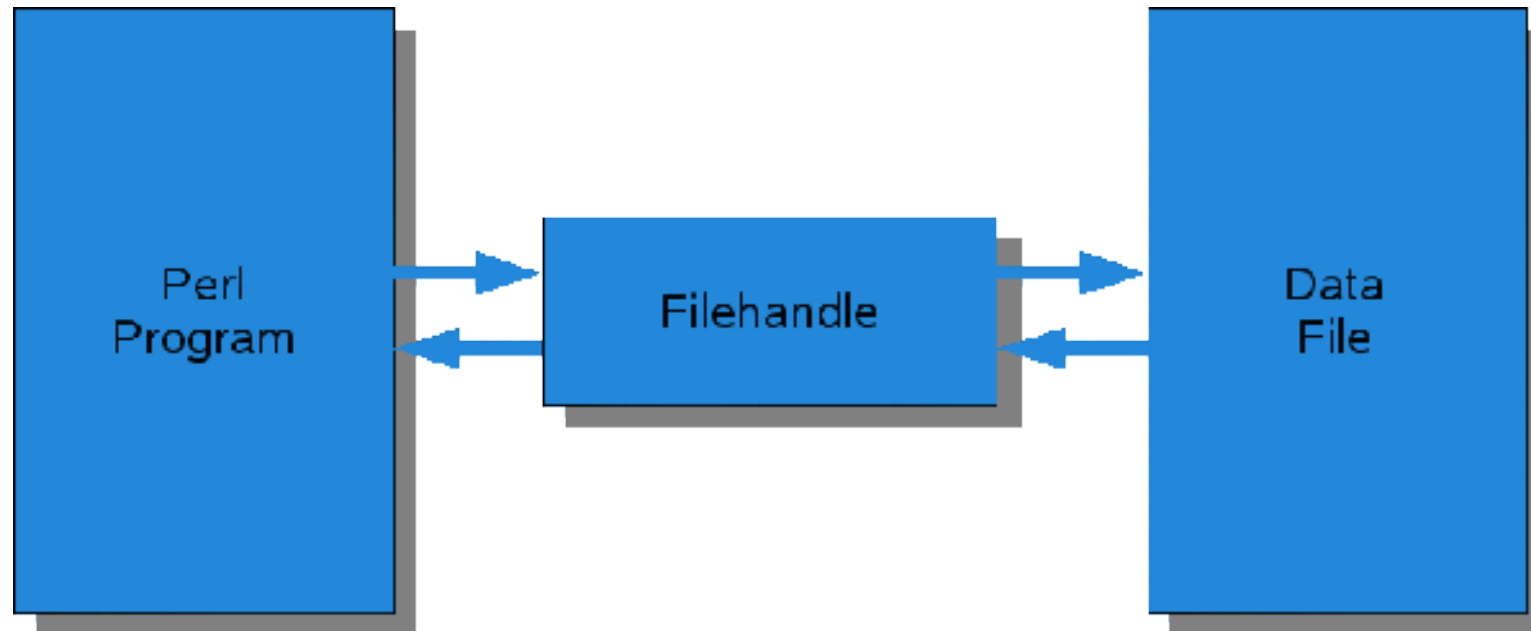
A filehandle in a Perl program is the name for an I/O connection between a Perl process and the outside world.

STDIN is a filehandle, naming the connection between the Perl process and the standard input. STDOUT (standard output) and STDERR (standard error output) are other filehandles.



It's customary to CAPITALIZE the names of filehandles.

# What is a Filehandle



In other words: A filehandle can be seen as a nickname for the files used in the PERL script.

A handle is a temporary name assigned to a file. A good choice for filehandle (name) is an abbreviated version of the filename.



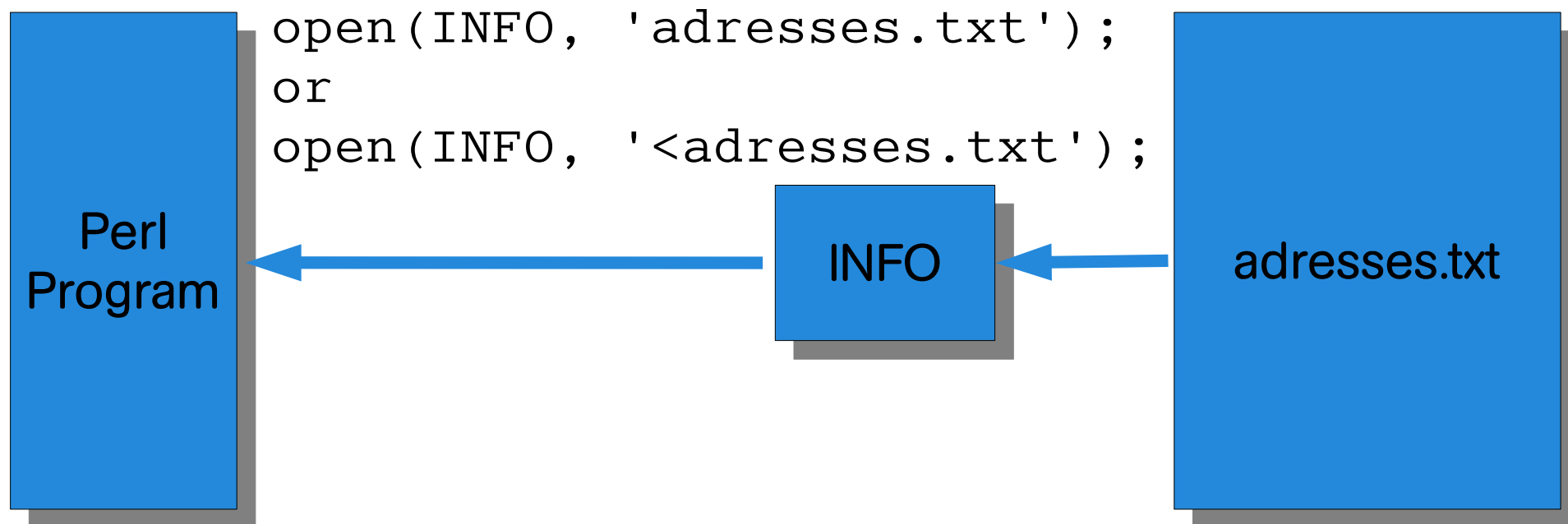
# Opening and Closing

Filehandles have to be opened with the `open()`-statement, except for `STDIN`, `STDOUT` and `STDERR`, which are automatically opened.

```
$file = '/home/homer/addresses.txt';  
open(INFO, $file);           # Open the file
```

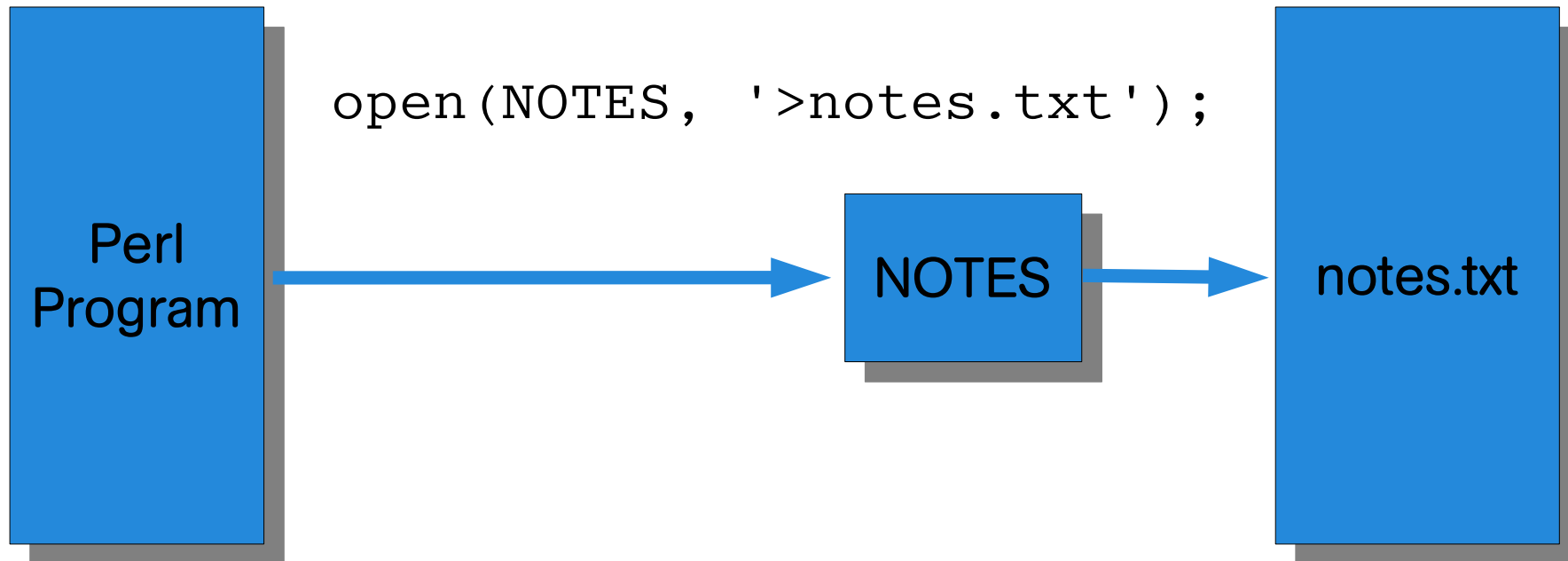
# Opening for reading

Open a file for reading:



# Opening for writing

Open a file for writing:

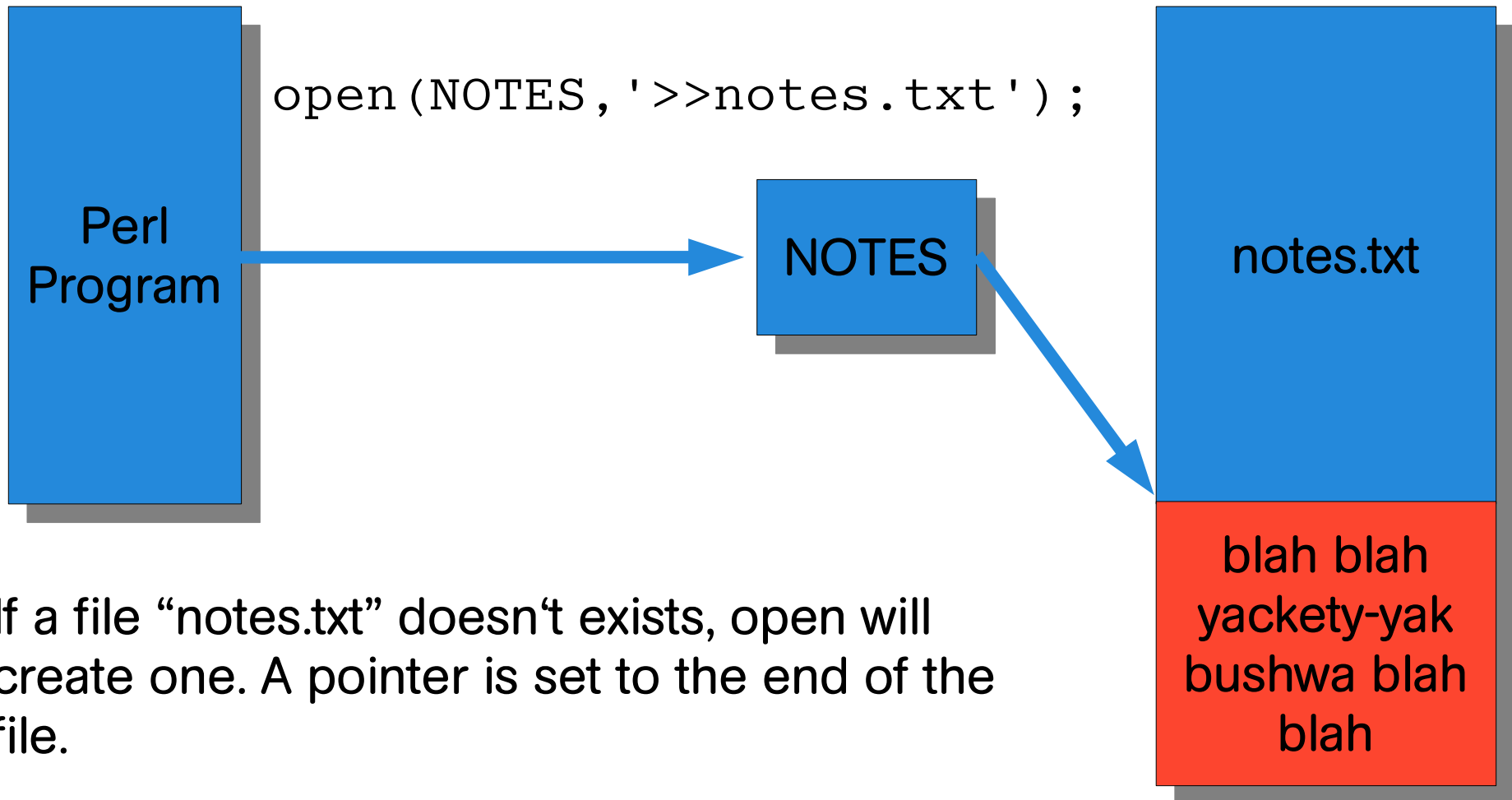


If a file “notes.txt” already exists, it will be overwritten.

**All existing data will be lost!**

# Opening for writing

Append to a file:



If a file “notes.txt” doesn’t exist, open will create one. A pointer is set to the end of the file.

# Weird things about open

Parenthesis can be omitted, i.e.

```
open INFO, $file;
```

The filename can be dropped as well, if a scalar variable with the same name as the filehandle exists:

```
$INFO = '<addresses.txt';  
open INFO ;  
@lines = <INFO>;  
close(INFO);  
print @lines;
```

# Problems with open

Every forms of open returns true for **success** and false for **failure**.

## Reasons for failure:

Opening a file for input may fail, if  
the file doesn't exist or  
cannot be accessed because of permissions;

Opening a file for output may fail, if the file is  
write-protected, or if  
the directory is not writable or accessible.

# The `close` Operator

If a script doesn't need the access to a file anymore, the file can (may) be closed by the `close` operator:

```
close(FILEHANDLE) ;
```

Reopening a filehandle also closes the previously opened file automatically.

## open the deeper way

So far, we had a look at the “open à la C”, it's closest to shell.

sysopen is the command for those who need a finer precision the C's fopen().

sysopen is a direct hook into the open system call.

```
sysopen HANDLE, PATH, FLAGS, [MASK]
```



# sysopen

```
sysopen HANDLE, PATH, FLAGS, [MASK]
```

HANDLE	argument is a filehandle just as with open PATH a literal path without greater-thans or less-thans or pipes or minuses
FLAGS	contains one or more values derived from the Fcntl module that have been or'd together using the bitwise " " operator.
MASK	optional; if present, it is combined with the user's current unmask for the creation mode of the file. Usually omitted.

# sysopen, Constants from Fcntl

<code>O_RDONLY</code>	Read only
<code>O_WRONLY</code>	Write only
<code>O_RDWR</code>	Read and write
<code>O_CREAT</code>	Create the file if it doesn't exist
<code>O_EXCL</code>	Fail if the file already exists
<code>O_APPEND</code>	Append to the file
<code>O_TRUNC</code>	Truncate the file
<code>O_NONBLOCK</code>	Non-blocking access

# Examples with sysopen

Open a file for writing:

```
open(FH, "> $path");
```

corresponding sysopen:

```
sysopen(FH, $path, O_WRONLY | O_TRUNC | O_CREAT);
```

Open a file for appending:

```
open(FH, ">> $path");
```

corresponding sysopen:

```
sysopen(FH, $path, O_WRONLY | O_APPEND | O_CREAT);
```

# Problems with `open`

A filehandle that couldn't be opened can still be used in the program without a warning.

If you read from the filehandle, you'll get end-of-file right away.

If you write to the filehandle, the data is silently discarded.

Ugly solution:

```
unless (open (INFO, ">addresses.txt")) {  
    print "I couldn't create addresses.txt\n";  
} else {  
    # the rest of your program  
}
```

# The Perfect Solution

# die

The `die` function takes a list within optional parentheses, gives out that list on the standard error output, and then ends the Perl process (program) with a nonzero exit status

```
unless (open (INFO, ">addresses.txt")) {  
    die "I couldn't create addresses.txt\n";  
}  
# the rest of the program
```

# Open that File or Die

```
open ( INFO, ">addresses.txt" ) ||  
    die "I couldn't create addresses.txt\n";  
  
# the rest of the program
```

The `||` (logical-or) makes sure that the `die` command gets executed, only when the result of `open` is false.

# Read Lines from a File

Once a filehandle is open for reading, you can read lines from it the way you read from standard input with STDIN

```
$file = '/home/homer/addresses.txt';  
open(INFO, $file);           # Open the file  
@lines = <INFO>;             # Read it into an array  
close(INFO);                 # Close the file  
print @lines;                # Print the array
```

another way to do it:

```
open (INFO, "/home/homer/addresses.txt") ;  
while (<INFO>) {  
    chomp;  
    print "$_\n";  
}
```

# Write to a File

If a filehandle is open for writing (or appending), you can print to it by using the print command immediately followed by the filehandle before the other arguments.

```
print RESULTS "The value is: $n\n";
```



## Exercise: CopyCat

Write a Perl script to copy data from one file into another file.

# Program: Copy File to File

```
#!/usr/bin/perl/

$in_file = "addresses.txt";
$out_file = ">addresses2.txt";

open(IN,$in_file) || die "cannot open $in_file
for reading: $!";
open(OUT,$out_file) || die "cannot create
$out_file: $!";
while (<IN>) {          # read a line from file $a
    into $_
    print OUT $_;      # print that line to file $b
}
close(IN) || die "can't close $in_file: $!";
close(OUT) || die "can't close $out_file: $!";
```

## (just) to be on the safe side ...

It's easy to accidentally overwrite some existing file.

There is a command to test if a file exists. With `-e $filevar` you can check if a file exists.

```
$in_file = "addresses.txt";  
if (-e $in_file) {  
    print "The file $in_file already exists";  
} else {  
    print "The file $in_file doesn't exist";  
}
```

# Other file tests

- r File or directory is readable
- w File or directory is writable
- x File or directory is executable
- o File or directory is owned by user
- e File or directory exists
- z File exists and has zero size (directories are never empty)
- s File or directory exists and has nonzero size  
(the value is the size in bytes)
- f Entry is a plain file
- d Entry is a directory
- l Entry is a symlink
- S Entry is a socket
- p Entry is a named pipe (a "fifo")
- b Entry is a block-special file (like a mountable disk)
- c Entry is a character-special file (like an I/O device)
- k File or directory has the sticky bit set
- T File is "text"
- B File is "binary"
- M Modification age in days
- A Access age in days
- C Inode-modification age in days

# The Gory Details with stat

You want the whole “story” of a file?

```
# check if STDIN is interactive and prompt if it is
print "File name? " if (-t STDIN);
chop ($name = <STDIN>);
```

```
@file_data = stat($name);
@description = ("device", "inode", "mode", "links",
"user id", "group id", "device id", "size",
"accessed", "modified", "changed", "blocksize",
"block count");
```

```
foreach $index (0 .. $#description) {
    printf "%-12s", $description[$index];
    print $file_data[$index], "\n";
}
```

# Difference: `stat()` and `lstat()`

If the argument is not a symbolic link `stat()` and `lstat()` supply the same results.

If you invoke the `stat` function on the name of a symbolic link, it will return information about the file the symbolic link points at (if accessible) and not information on the symbolic link itself.

`lstat` provides information about the symbolic link itself.

# Exercises

- 1) Write a program to read a file and output every line preceded with a line number to another file
- 2) Write a program to read in a list of filenames and then display which of the files are readable, writable, and/or executable, and which ones don't exist.

# Reading Multiple Files

If no file handle is used with the diamond operator, Perl will check the `@ARGV` variable. If `@ARGV` is empty, the diamond operator will read from `STDIN`, i.e. from keyboard or from a redirected file.

```
while (<>) {  
    print();  
}
```

If called with

```
multiple_file_read.pl abc.txt efg.txt
```

the content of `abc.txt` followed by `efg.txt` will be printed.



# Directory Access

The `chdir` function takes a single argument - an expression evaluating to a directory name to which the current directory will be set.

`chdir` returns `true` when the script was able to change to the requested directory and `false` if it wasn't.

```
chdir("/opt") || die "cannot cd to /opt ($!)";
```



**Parenthesis are optional**

## “Strange” usages of chdir

```
if ( chdir "/opt" ) {  
    print "We got there!";  
} else {  
    print "We go to tmp instead!";  
    chdir /tmp;  
}
```

# Globbering

The expansion of arguments like `*` or `/home/homer/don*` into a list of matching filenames is called **globbing**.

To invoke globbing the pattern has to be put between angle brackets or has to be the argument of the `glob` function.

```
@a = </opt/kde*>;  
@a = glob( "/etc/*ca*" );
```

# Directory Handles

```
opendir DIRHANDLE,EXPR
```

`opendir` opens a directory named `EXPR` for processing by `"readdir"`, `"telldir"`, `"seekdir"`, `"rewinddir"`, and `"closedir"`. It returns `true` if successful. `DIRHANDLE` may be an expression whose value can be used as an indirect dirhandle, usually the real dirhandle name.

Dirhandles have their own namespace separate from Filehandles.

# Reading a Directory Handle

When a directory handle is open, we can read the list of (file)names with `readdir`, which takes a single parameter: the directory handle.

Each invocation of `readdir` in a scalar context returns the next filename (basename) in a random order.

If there are no more names, `readdir` returns `undef`.

If `readdir` is invoked in a list context all the names are supplied as a list.

# readdir example

```
my $dir = '.';
opendir(DIR, $dir) or die $!;

while (my $file = readdir(DIR)) {
    # Use a regular expression to ignore
    # files beginning with a #
    next if ($file =~ /^[#]/);
    print "$file\n";
}

closedir(DIR);
exit 0;
```



# Return value of `unlink`

The return value of `unlink` is the number of files successfully deleted.

If this number is equal to the number of files in the filelist given to `unlink`, everything is fine. If this number is smaller, you are faced with the question, which files couldn't be deleted.

So stepping through the list might be better than the previous approach:

```
foreach $file (<*~>)  
    unlink($file) || warn "having trouble deleting  
$file: $!";  
}
```



# Renaming

Another often needed function is the renaming of files.

It's easy to accomplish:

```
rename($old_file, $new_file);
```

Of course, rename should be “guarded”:

```
rename(Raider, Twix) || die "Renaming failed"
```

## rename: difference to mv

In Unix/Linux the following commands are equivalent:

```
mv twix /opt/comp/                    and  
mv twix /opt/comp/twix
```

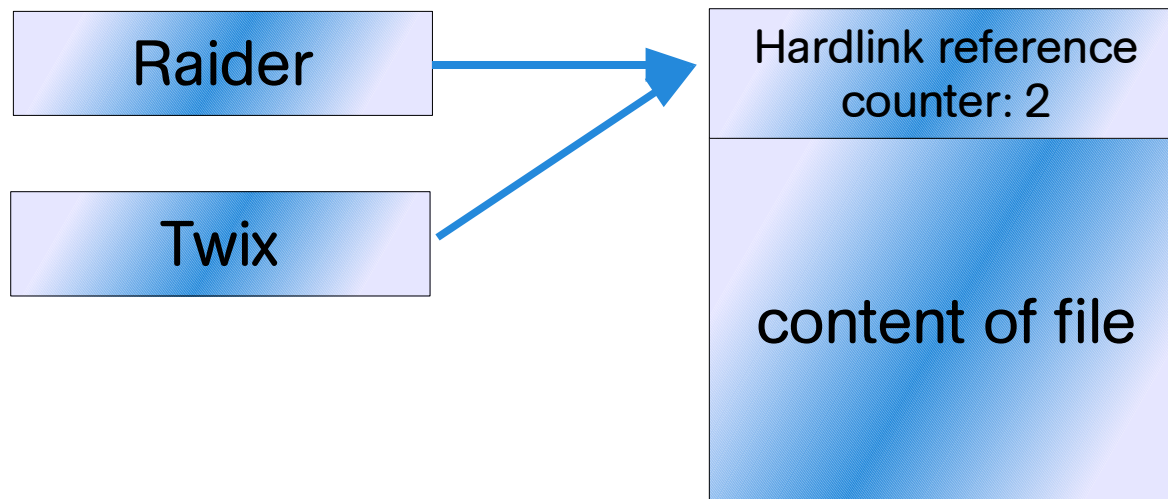
Whereas in Perl the target filename has always to be given explicitly:

```
rename( "twix" , "/opt/comp/twix" );
```

# Hard and Soft Links

A hard link is indistinguishable from the original file.

The references to a file are counted.



# Restriction for Hardlinks

A hard link to a file must reside on the same mounted filesystem.

A hard link for a directory is not possible, because the filesystem is strictly hierarchically organized. Allowing links to directories would inevitably lead to a mish-mash.

# Symbolic Links

Also called “soft links”

If a symbolic link is used in a Perl command, the corresponding linked file is used instead.

A symbolic link is a special file containing a pathname as data.

The contents of symlinks don't have to refer (point to) existing files or directories.

Chains of symbolic links are possible.

# Creating Hard and Soft Links

```
link($old_filename, $new_filename)
```

A hardlink from the file `$old_filename` to `$new_filename`.  
`$old_filename` must exist!

An example:

```
link("raider", "twix")  
    || die "cannot link raider to twix";
```

Symbolic links are created in Perl with the `symlink` command:

```
symlink("raider", "twix")  
    || die "cannot link raider to twix";
```

Now “raider” doesn’t have to exist and “twix” can be on a different filesystem.

# Readlink

```
readlink( EXPR )
```

```
readlink EXPR
```

Returns the value of a symbolic link, if symbolic links are implemented. If not, gives a fatal error. If there is some system error, returns the undefined value and sets \$! (errno). If EXPR is omitted, it uses \$\_.

```
if (defined($x = readlink("twix"))) {  
    print "twix points at '$x'\n";  
}
```

# Making Directories

```
mkdir (FILENAME, MODE)
```

Creates the directory specified by `FILENAME`, with permissions specified by `MODE`. If it succeeds it returns 1, otherwise it returns 0 and sets `$! (errno)`.

```
mkdir("sweets", 0755) || die "cannot mkdir  
sweets: $!";
```



# Removing Directories

```
rmmdir(FILENAME)
```

```
rmmdir FILENAME
```

Deletes the directory specified by `FILENAME` if it is empty. If it succeeds it returns 1, otherwise it returns 0 and sets `$!(errno)`. If `FILENAME` is omitted, uses `$_`.

```
rmmdir("sweets") || die "cannot rmmdir sweets:$!";
```

# Linux/UNIX vs. Perl File Permissions



# ls -l

```
jupiter:/home/Debra > ls -l
```

```
total 92
```

```
drwxr-xr-x    2 bernd    users          35 2003-10-25 11:43 Dokumente
drwxr-xr-x    2 bernd    users          35 2003-10-25 11:44 Mail
drwxr-xr-x    7 bernd    users        287 2003-10-25 13:18 Kursunterlagen
-rw-r--r--    1 bernd    users        187 2003-11-10 19:36 zitat.txt
-rw-r--r--    1 bernd    users    46992 2003-11-11 19:09 shell.jpg
-rwxr--r--    1 bernd    users    42288 2003-11-11 19:09 pipe.gif
```

```
jupiter:/home/Debra >
```

1	2	3	4	5	6	7	8	9	10
Type	User Permissions			Group Permissions			Other Permissions		
Type	Read	Write	Execute	Read	Write	Execute	Read	Write	Execute
d	r	w	x	r	-	x	r	-	x

4

2

1

# Modifying Permissions

Under Linux/UNIX permissions on a file are changed with the `chmod` command.

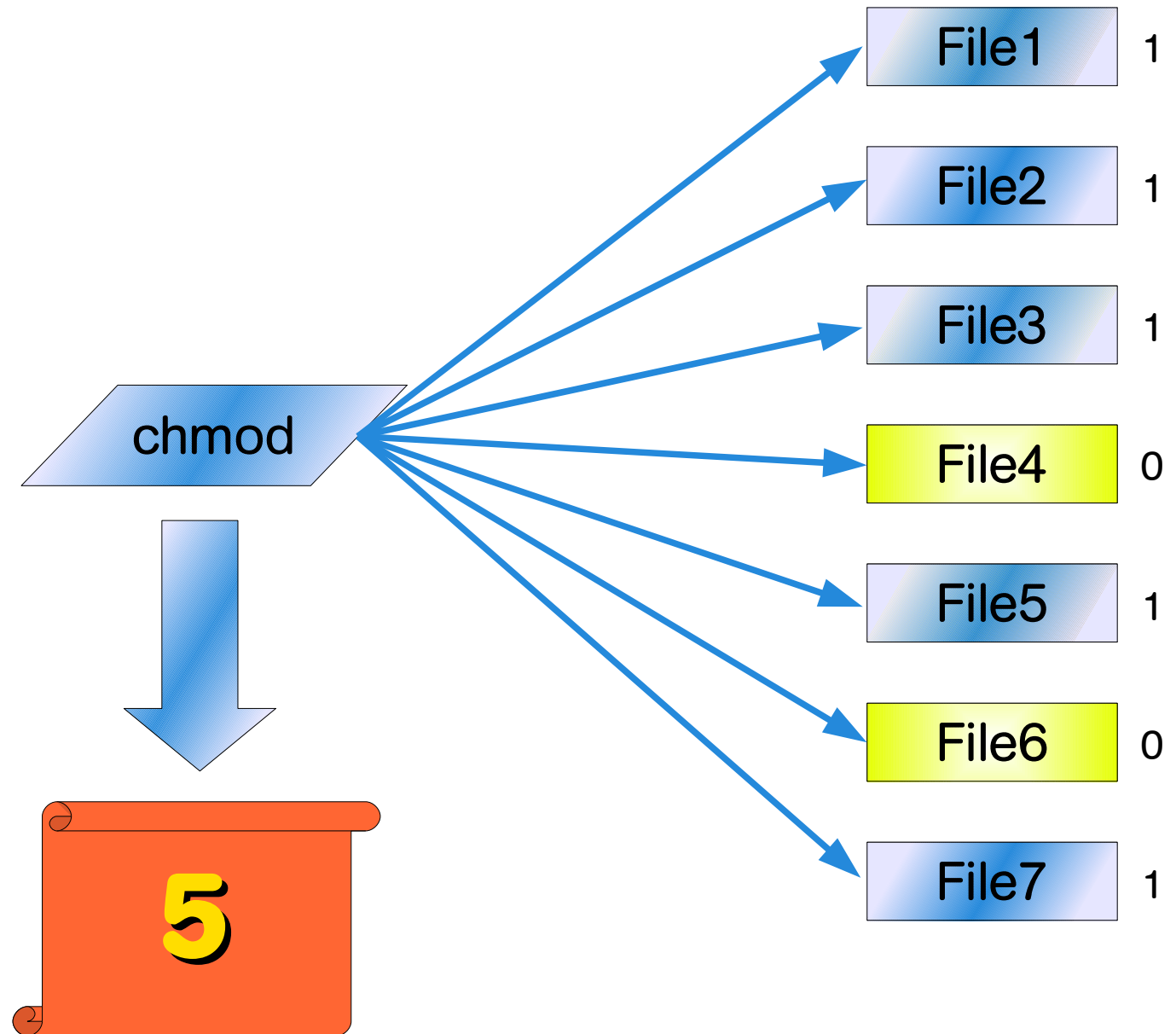
Similarly, Perl changes permissions using a function with the same name.

The permissions on a file or directory define who can do what to that file or directory.

The `chmod` function takes an octal numeric number as the mode and a list of filenames. “Perl” tries to alter the permissions of all the filenames to the indicated mode.

```
chmod( 0644 , "homer" , "lisa" , "bart" , "marge" ) ;
```

# chmod



# Return value of chmod

The return value of chmod is the number of files successfully adjusted, regardless if the adjustment is different to the previous setting.

```
foreach $file ( "homer", "marge" ) {  
    unless chmod (0644,$file) {  
        warn "Couldn't chmod $file: $!";  
    }  
}
```



# Change Ownership

# chown

Every file, directory, device or whatever in the filesystem has an owner and belongs to a group.

The owner and group of a file determine to whom the owner and group permissions, i.e. read, write, and/or execute, apply .

The owner and group of a file are set with the creation of the file, but in many cases it's necessary to change them later on.

The Perl command to accomplish this:

```
chown LIST
```

The LIST consists of UID, GID and a list of files to be changed

```
chown(1004, 4711, "apples", "oranges", "bananas" );
```