

Andrew Shitov

Using Perl 6

*100 Programming Challenges
Solved with the Perl 6
Programming Language*

DeepText — 2017

Using Perl 6

*100 Programming Challenges Solved with
the Perl 6 Programming Language*

© Andrew Shitov, author, 2017

This book is a collection of different programming challenges and solutions in Perl 6. It can be used as an exercise book, when you are learning Perl 6, or as a reference book when you are teaching it.

It is assumed that the reader knows the basics of Perl 6 and wants to have some practice.

1st edition published on 1 November 2017

Published by DeepText, Amsterdam
www.deeptext.media

ISBN 978-90-821568-1-2

Contents

Foreword	10
Prerequisites	11
How to debug programs.....	12

Part 1

Chapter 1. Strings

1.1. Using strings

1. Hello, World!.....	18
2. Greet a person	19
3. String length.....	20
4. Unique digits.....	22

1.2. Modifying string data

5. Reverse a string	24
6. Removing blanks from a string	25
7. Camel case	26
8. Incrementing filenames	27
9. Random passwords.....	28
10. DNA-to-RNA transcription.....	30
11. Caesar cipher	31

1.3. Text analysis

12. Plural endings.....	34
13. The most frequent word	35
14. The longest common substring	36
15. Anagram test	37
16. Palindrome test	38
17. The longest palindrome	40
18. Finding duplicate texts	42

Chapter 2. Numbers

2.1. Using numbers

19. π	48
20. Factorial!	49
21. Fibonacci numbers	52
22. Print squares	53
23. Powers of two	54
24. Odd and even numbers	55
25. Compare numbers approximately	56
26. Multiplying big numbers	57
27. Prime numbers.....	58
28. List of prime numbers.....	59
29. Prime factors.....	60
30. Reducing a fraction	61
31. Divide by zero	62

2.2. Random numbers	
32. Generating random numbers.....	64
33. Neumann's random generator	65
34. Histogram of random numbers.....	67
2.3. Mathematical problems	
35. Distance between two points	70
36. Standard deviation	71
37. Polar coordinates.....	73
38. Monte Carlo method.....	75
2.4. Numbers and strings	
39. Unicode digits	78
40. Guess the number.....	79
41. Binary to integer.....	80
42. Integer as binary, octal, and hex	81
43. Sum of digits.....	82
44. Bit counter	83
45. Compose the largest number	85
46. Convert to Roman numerals	86
47. Spelling numbers.....	88

Chapter 3. Aggregate Data Types

3.1. Manipulating lists and arrays

48. Swap two values.....	94
49. Reverse a list	95
50. Rotate a list	96
51. Randomise an array	97
52. Incrementing array elements	98
53. Adding up two arrays	99
54. Exclusion of two arrays	101

3.2. Information retrieval

55. Sum of the elements of an array.....	104
56. Average of an array.....	105
57. Moving average.....	106
58. Is an element in a list?	107
59. First odd number	108
60. Take every second element	109
61. Number of occurrences in array	110
62. Finding unique elements	111
63. Minimum and maximum.....	112
64. Increasing sequences	113

3.3. Working with subroutines

65. Passing arrays to subroutines	116
66. Variadic parameters in a sub	117

3.4. Multi-dimensional data	
67. Transpose a matrix	120
68. Sort hashes by parameter	121
69. Count hash values	122
70. Product table.....	123
71. Pascal triangle.....	125

Chapter 4. Regexes and Grammars

4.1. Regex matching	
72. Count vowels in a word.....	130
73. Count words	131
74. Skipping Pod documentation	132
75. Currency converter	133
4.2. Substitutions with regexes	
76. Double each character.....	136
77. Remove duplicated words	137
78. Separate digits and letters	138
79. Separate groups of digits	139
80. Increase digits by one	140
81. Pig Latin.....	141
82. Simple string compressor	142
83. %Templating% engine	143

4.3. Using grammars	
84. Decode Roman numerals	146
85. Balanced parentheses	148
86. Basic calculator.....	150

Part 2

Chapter 5. Date and Time

87. Current date and time.....	156
88. Formatted date	157
89. Datetime arithmetic.....	158
90. Leap years	159

Chapter 6. Parallel Computing

91. Setting timeouts.....	162
92. Sleep Sort	164
93. Atomic operations.....	166
94. Parallel file processing	168

Chapter 7. Miscellaneous

95. The <code>cat</code> utility	172
96. The <code>uniq</code> utility	173
97. Reading directory content	174
98. Text to Morse code.....	175

99. Morse to text	177
100. Brainfuck interpreter.....	178
What's next?	183

Foreword

In this book, you will find 100 solutions of different common programming challenges that are written in Perl 6. Once you already know some of the basics of the language, it is a good idea to spend some time trying to do some exercises and the most popular tasks, which will vary from displaying the *Hello, World!* greeting, to creating a parser for the command-line calculator.

The seven chapters are divided into two parts, which cover the essential parts of Perl; firstly, the strings, numbers, and aggregate data structures, and secondly, of Perl 6 in particular, namely, the new regexes, grammars, and parallel computing.

Each of the 100 tasks presented in the book demonstrates the main idea of how to approach a problem; big chunks of code are avoided, as much as possible.

However, you are invited not to limit yourself to the proposed solution, though. First, try other approaches for the initial task, and then, extend the task with additional conditions and solve it. Do as many exercises in Perl 6 as possible, because it will make you more fluent in this great language.

Chapter 6

Parallel Computing

91. Setting timeouts

Do not wait for a slow code block if it takes too long.

In Perl 6, promises are the best way to create timeouts. In the following example, two code blocks are created; they are executed in parallel.

```
my $timeout = Promise.in(2).then({
    say 'Timeout after 2 seconds';
});

my $code = start {
    sleep 5;

    say 'Done after 5 seconds';
}
```

Both `$timeout` and `$code` are the promises, i. e. objects of the `Promise` type. Actually, the `$timeout` variable is a promise that is executed as a result of keeping the anonymous promise created by the `Promise.in(2)` call. The `in` method of the `Promise` class creates a promise that becomes kept after the given number of seconds.

The second promise, stored in the `$code` variable, is created by the `start` function. This is a long-running code block, which does not return within five seconds. The `$code` promise can be kept only after that time.

The `$timeout` promise is kept earlier than the `$code` one. To let the program continue earlier, create another promise with the help of the `anyof` method:

```
await Promise.anyof($timeout, $code);

say 'All done';
```

The flow of the whole program is the following: first, the `$timeout` code block is created and starts running in a separate thread. Then, without waiting, the `$code` block has been created and launched. Finally, the next line of the main thread is executed; it creates an anonymous thread and waits until it is kept. The `await` routine blocks the execution of the main program until at least one of its arguments is kept. This program prints 'All done' after two seconds, and exits:

```
$ perl6 timeout.pl
Timeout after 2 seconds
All done
```

If the `$code` thread is completed first (say, if we changed the timeout value to 10 seconds), then the output is different, and the timeout is not triggered:

```
$ perl6 timeout.pl
Done after 5 seconds
All done
```

Keep in mind that in our examples, the program finishes after printing 'All done'. In case the program continues after that, the longest promise will still be running.

For example, add a simple delay like the one shown below:

```
await Promise.anyof($timeout, $code);
say 'All done';
sleep 20;
```

In this case, the program prints all three messages:

```
$ perl6 timeout.pl
Timeout after 2 seconds
All done
Done after 5 seconds
```

92. Sleep Sort

Implement the Sleep Sort algorithm for a few small positive integer values.

The *Sleep Sort* is a funny implementation of the sorting algorithm. For each input value, an asynchronous thread starts, which waits for the number of seconds equals to the input number and then prints it. So, if all the threads are spawned simultaneously, the output of the program contains the sorted list.

Here is the solution in Perl 6. On the next page, we will go through the bits of it and explain all the important moments.

```
await gather for @*ARGS -> $value {
  take start {
    sleep $value/10;
    say $value;
  }
}
```

Pass the values via the command line, and get them sorted.

```
$ perl6 sleep-sort.pl 9 10 2 8 5 7 6 4 1 3
1
2
3
. . .
8
9
10
```

The input values from the command line come to the @*ARGS array. The first step is to iterate over the array:

```

for @*ARGS -> $value {
    . . .
}

```

For each `$value`, the `start` block creates a promise with a code block that waits for the time that is proportional to the value and prints the value after that time.

```

start {
    sleep $value/10;
    say $value;
}

```

Dividing the value by ten speeds up the program. On the other hand, the delay should not be too small to avoid race conditions between different threads.

After a separate promise has been created for each input number, the program has to wait until all of them are kept. To achieve that, the `gather`—`take` construction is used. The `take` keyword adds another promise to a sequence, which is then returned as a whole by the `gather` keyword.

```

gather for @*ARGS -> $value {
    take start {
        . . .
    }
}

```

Finally, the `await` routine ensures the program does not quit until all the promises are kept or, in other words, until all the numbers are printed.

```

await gather . . . {
    take start {
        . . .
    }
}

```

93. Atomic operations

Using the solution for Task 38, the Monte Carlo method, create a program that calculates the result using multiple threads.

Perl 6 has built-in support for parallel computing. In Task 92, *Sleep Sort*, we've seen how to use the keywords `await`, `gather`, and `take` to spawn a few threads and wait for them to finish.

When different threads want to modify the same variable, such as a counter, it is wise to introduce atomic operations to make sure the threads do not interfere with each other. Here is the modification of the Monte Carlo program calculating the area of a circle with four parallel threads.

```
my atomicint $inside = 0;

my $n = 5000;
my $p = 4;

await gather for 1..$p {
  take start {
    for 1..$n {
      my @point = map {2.rand - 1}, 1..2;
      $inside⚡++ if sqrt([+] map **2, @point) <= 1;
    }
  }
}

say 4 * $inside / $p / $n;
```

Run the program a few times, changing the value of `$n` (the number of random points per thread) and `$p` (the number of threads). The program should print the value that is close to π , such as `3.141524`.

The new thing here is the atomic increment operation:

```
$inside⚡++
```

An atomic operation ensures that the variable is modified with no conflicts between the threads.

The variable itself should be a native integer of a special type—notice how it is declared:

```
my atomicint $inside;
```

As the atomic operation uses the Unicode character, there is an ASCII alternative:

```
atomic-fetch-inc($inside)
```

Here's a list of other atomic operations and their synonyms that can be used with parallel processes:

<code>\$var⚡= \$value</code>	<code>atomic-assign(\$var, \$value)</code>
<code>my \$a = ⚡\$var</code>	<code>my \$a = atomic-fetch(\$var)</code>
<code>\$var⚡++</code>	<code>atomic-fetch-inc(\$var)</code>
<code>\$var⚡--</code>	<code>atomic-fetch-dec(\$var)</code>
<code>++⚡\$var</code>	<code>atomic-inc-fetch(\$var)</code>
<code>--⚡\$var</code>	<code>atomic-dec-fetch(\$var)</code>
<code>\$var⚡+= \$value</code>	<code>atomic-fetch-add(\$var, \$value)</code>
<code>\$var⚡-= \$value</code>	<code>atomic-fetch-dec(\$var, \$value)</code>

N. B. The code in this task works with the Rakudo Perl 6 compiler starting from version 2017.09. Earlier versions do not support atomic operators.

94. Parallel file processing

Process the files from the current directory in a few parallel threads.

We have to do something with each file in the directory, and it has to be done in such a way that files are processed independently with a few workers. It is not possible to predict how long the process will take for each individual file, that's why we need a common queue, which supplies the file-names for the next available worker.

A good candidate for the queue is a channel.

```
my $channel = Channel.new();
$channel.send($_) for dir();
$channel.close;
```

All the file names are sent to the channel, which we close afterward. (On how to read directories, see more details in Task 97, *Reading directory contents*.)

Channels are designed to work thread-safe. It means that it is possible to get data from the channel using several threads, and each value is processed only once. Perl 6 cannot predict which thread gets which name but it can guarantee that each data item is only read by the threads once.

```
my @workers;
for 1..4 {
    push @workers, start {
        while (my $file = $channel.poll) {
            do_something($file);
        }
    }
}
```

The code on the previous page creates four independent workers using the `start` keyword. As they are executed independently not only from each other but also from the main program, it is important to wait until all of them are done:

```
await(@workers);
```

The elements of the `@workers` array are promises (objects of the `Promise` data type). The `await` routine waits until all the promises are kept.

Another practical way of creating and waiting workers is shown in Task 92, *Sleep Sort*: instead of collecting them in an array, you can use the `gather` and `take` keywords.

Examine the main loop:

```
while (my $file = $channel.poll) {
    do_something($file);
}
```

On each iteration, a value from the channel is read. The `poll` method ensures that the reading stops after the channel is exhausted.

All four threads are doing similar work and are polling the same channel. This approach distributes the filenames that were sent to the channel between the workers. As a name has been read, it is removed from the channel, and the next read request returns the next name.

Finally, cook the `do_something` sub according to your needs. In the following simplest example, it only prints filenames:

```
sub do_something($file) {
    say $file.path;
}
```