ITCS 4102/5102 Term Project Fall 2013

Survey Of Programming Languages

Programming Language: Perl

Pratik Ramani | Hassan Alasiri | Xinghe Lu | Franklin Young

The paradigm of the language

- Perl, sometimes referred to as "Practical Extraction and Reporting Language", is a family of high-level, general-purpose, interpreted, dynamic programming languages. The languages in this family include Perl 5 and Perl 6.
- The perl is the **multi-paradigm**. It has following programming paradigm
  - Imperative programming,
  - object-oriented (class-based) programming,
  - reflective programming,
  - procedural programming,
  - data-driven programming,
  - generic programming.

Some historical account of the evolution of the language and its antecedents

Evolution Of *Perl* :

- Perl was introduced in 1987, when the author, Larry wall, released **Perl 1**. The reason for its creation was that Wall was unhappy by the functionality of sed, C, awk and the Bourne Shell offered him. He created perl which was influenced by the languages like AWK, Smalltalk 80, Lisp, C,C++, sed,Unix shell, Pascal and combine all their best features in the one language. Since then perl has seen serval versions, each adding additional functionality. The Perl was originally implemented using C.
- **Perl 2**, released in 1988 : featured a better regular expression engine.
- **Perl 3**, released in 1989 : add features for binary data streams.
- **Perl 4**, before 1991, there was only a single man page(short for **manual page**) documentation for perl. But in 1991, *Programming perl*, known as "Camel book"(because of its cover) was published and became the main reference for the perl. Because of that perl version number changed to 4. There wasn't any changes in the language but to identify the version that was well documented by the book.
- **Perl 5**, which was released in 1994, was a complete re-write of the perl interpreter, and introduced such things as hard references, modules, objects and lexical scoping. Perl 5 gained widespread popularity in the late 1990s as a CGI scripting language, in part due to its parsing abilities. Perl 5 is used for graphics programming, system administration, networking programming, finance, informatics and other application.
- There are more development and improvement in the perl 5. Perl 5.0 was released in 1994 and Last version of perl 5 that was perl 5.18 was released in May 18, 2013. Following graph is showing the evolution of perl 5.

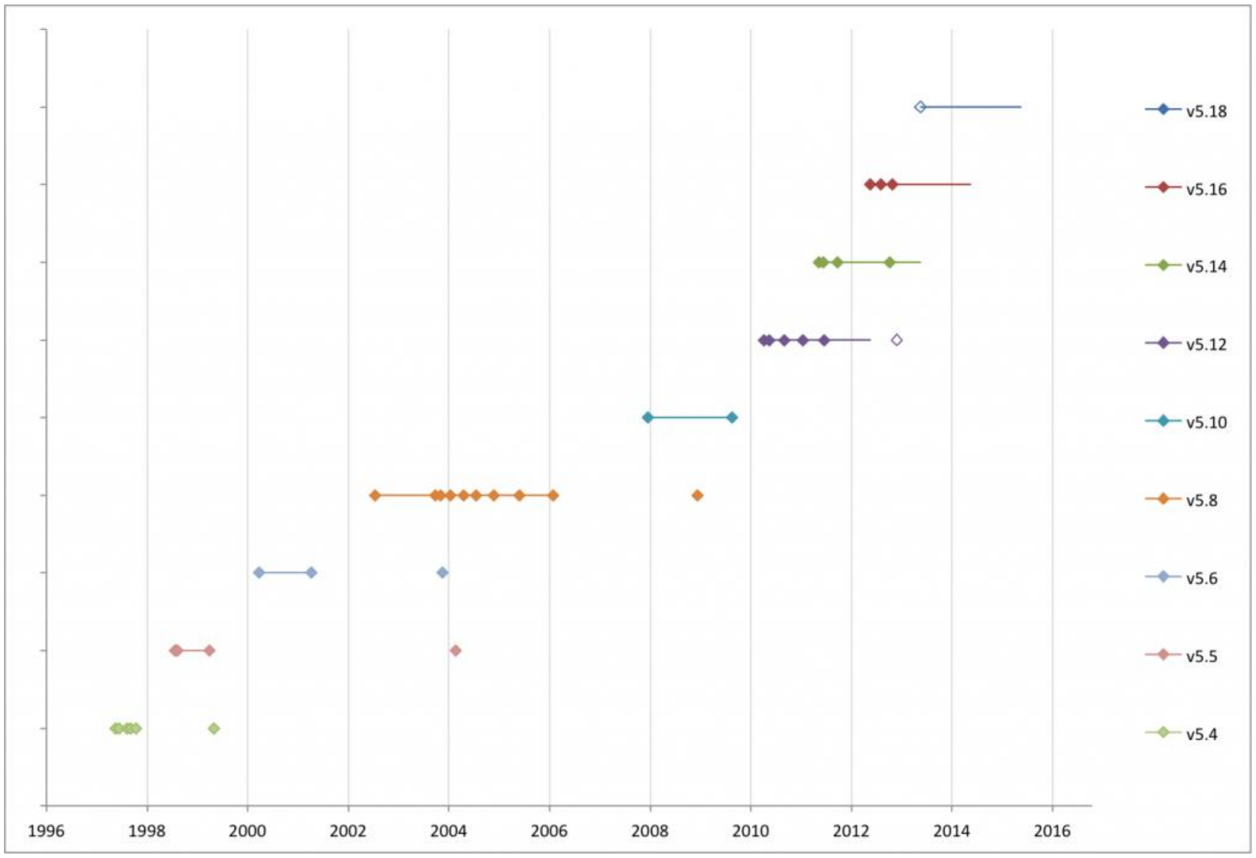


Fig1. Evolution of perl 5 (From "http://www.dagolden.com")

- Perl became especially popular as a language for writing server-side scripts for web-servers. But that's not the only use of perl, as it is commonly used for system administration tasks, managing database data, as well as writing GUI applications.

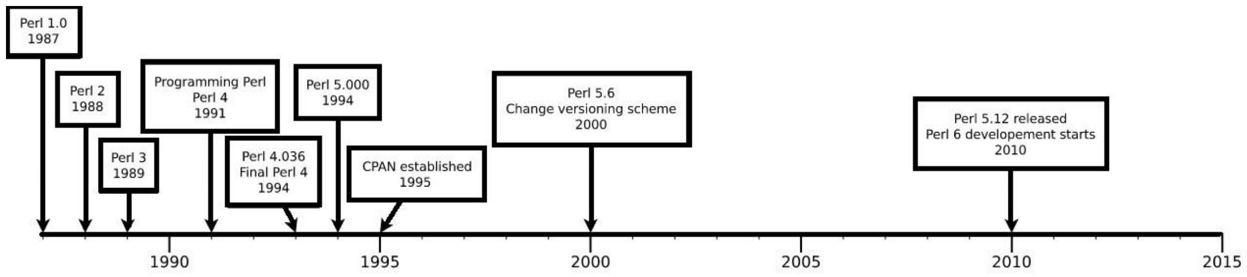

Perl 1.0 1987
Perl 2 1988
Perl 3 1989
Programming Perl Perl 4 1991
Perl 4.036 Final Perl 4 1994
Perl 5.000 1994
CPAN established 1995
Perl 5.6 Change versioning scheme 2000
Perl 5.12 released Perl 6 developement starts 2010
1990
1995
2000
2005
2010
2015

Fig2. Evolution of perl (From "http://commons.wikimedia.org/wiki/File:Perl_history.svg")

In the history of perl, it was known as following name due to its multiple features & use.

1. **"The Swiss Army chainsaw of scripting languages"** : because of its flexibility and power, and possibly also because of its perceived "ugliness"

2. **"Duct tape that holds the internet together "** : In 1998, Use of perl is ubiquitous as a glue language and perceived inelegance.

Languages like Python, PHP, Ruby, ECMAScript, LPC, Windows PowerShell, JavaScript, Falcon, Perl 6, Qore were influenced by perl.

Unlike other programming languages such as C and Lisp, which once created, remained the same, Perl has been improving for years and is still hungry for perfection, at least in terms of its capabilities. Currently, the next version of Perl, namely Perl 6, which began as a redesign of perl 5 in 2000, eventually evolved into a separate language, is in the making. No date has yet been set for its release.

The elements of the language: reserved words, primitive data types,structured types

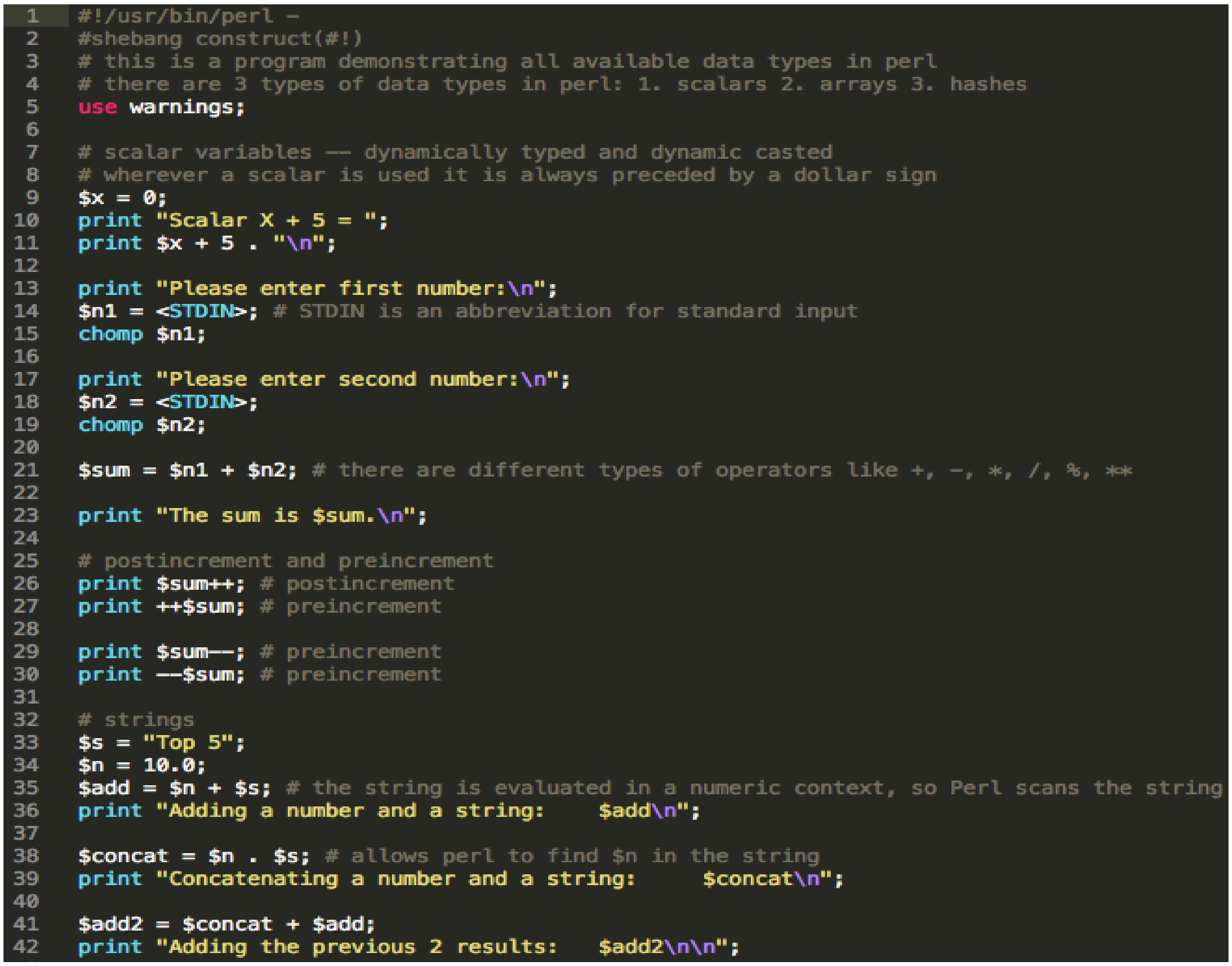
Reserved words in Perl include predefined variables and functions. Since regular variable names always start with a "$" or other non-alphabetical/numerical character, the reserved words do not conflict with user-defined variable names. However, words included in elements such as labels and filehandles do not follow a special character, thus lexical conflicts in Perl may still arise. Most reserved words are entirely lowercase to allow for better readability.

User-defined modules and classes should be named with initial capitals for to separate them from the built-in modules of Perl. The built-in modules are called Pragmas and each module has its own respective reserved words. For a full list, refer to the Pragma documentation located here: http://perldoc.perl.org/index-pragmas.html. Other reserved words can be found in the "perlvar" and "perlfunc" pages in the official Perl documentation.

Perl has three built-in data types, which are scalars, arrays of scalars, and hashes -- also called associative arrays of scalars. Scalars can be a number, a single string of any size, or a reference to something. Arrays are ordered lists of scalars similar to C++ and Java, where the lists are indexed by number and start with 0. Lastly, hashes are unordered collections of scalar values that are indexed by their associated string key.

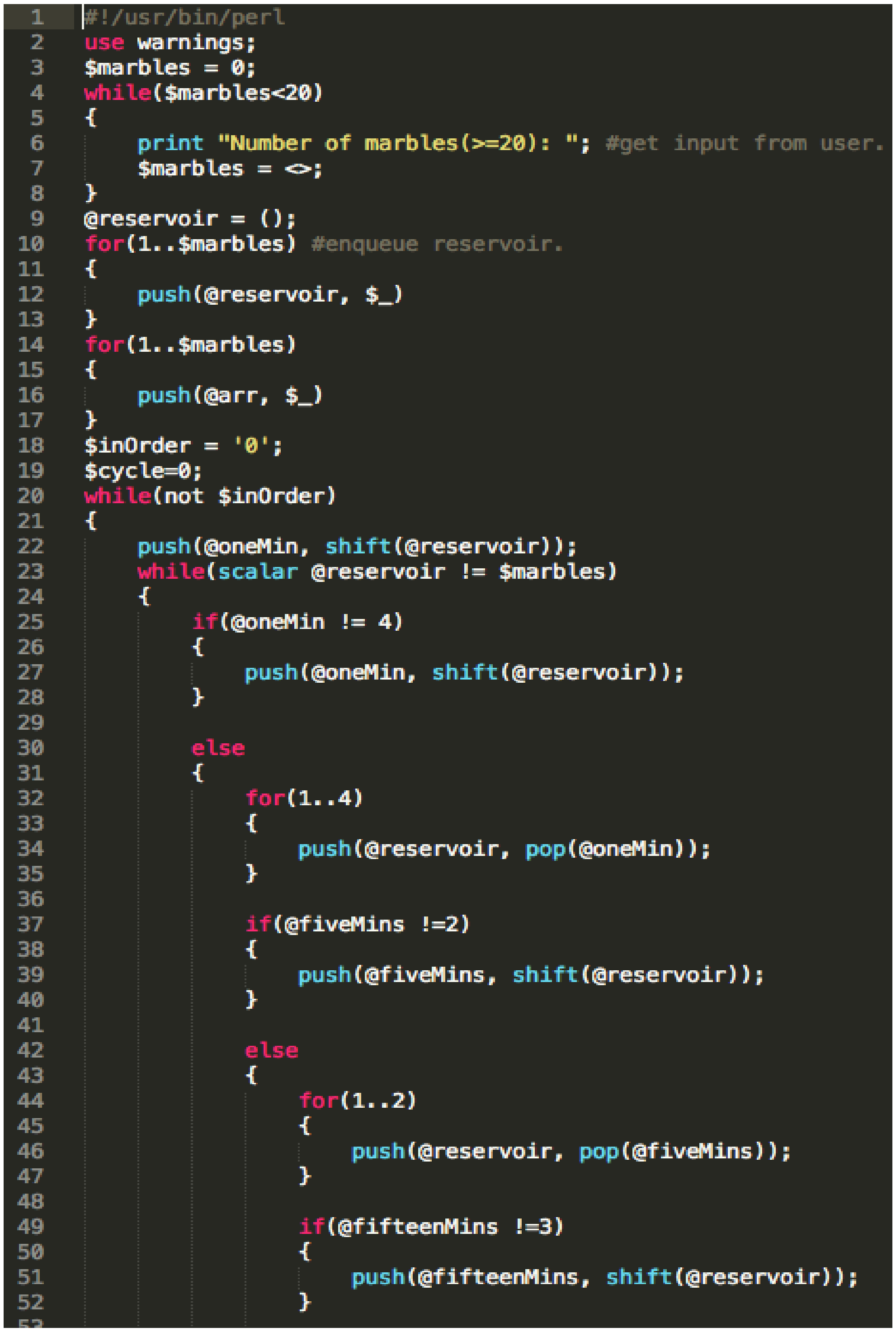
Judging by the definitions, one may assume that Perl is a limited language, however there are aspects of these data types that argue otherwise. For example, array size can be dynamically changed compared to the fixed size of arrays in Java and C++. This allows arrays to be treated like stacks and queues. Push, pop, and shift (de-queue) functions are embedded in Perl. Hashes are also easy to implement because Perl allows the programmer to avoid the creation of two arrays for comparisons via indexes. Examples of scalars, arrays, and hashes are shown below. In the array and hash examples, the push, pop, and shift functions are present.

The first example shows a basic scalar:



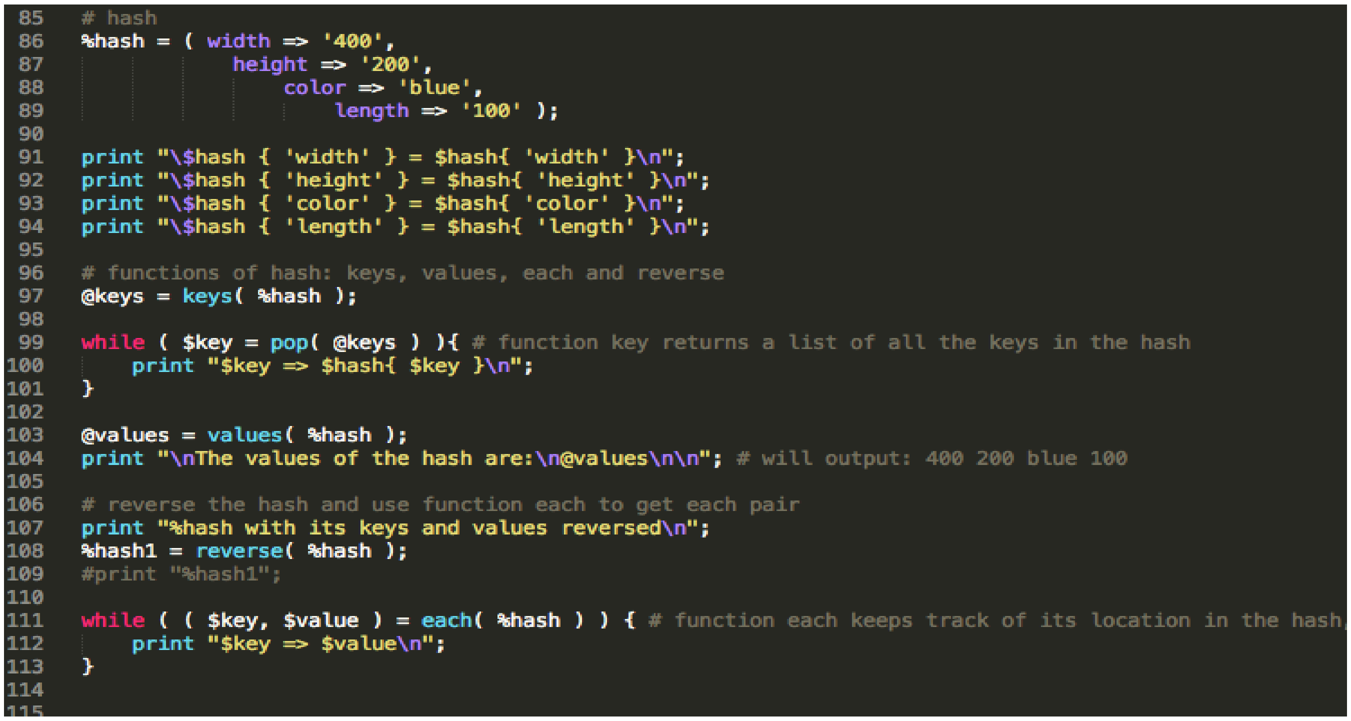
```perl
#!/usr/bin/perl -
#shebang construct(#!)
# this is a program demonstrating all available data types in perl
# there are 3 types of data types in perl: 1. scalars 2. arrays 3. hashes
use warnings;

# scalar variables -- dynamically typed and dynamic casted
# wherever a scalar is used it is always preceded by a dollar sign
$x = 0;
print "Scalar X + 5 = ";
print $x + 5 . "\n";

print "Please enter first number:\n";
$n1 = <STDIN>; # STDIN is an abbreviation for standard input
chomp $n1;

print "Please enter second number:\n";
$n2 = <STDIN>;
chomp $n2;

$sum = $n1 + $n2; # there are different types of operators like +, -, *, /, %, **

print "The sum is $sum.\n";

# postincrement and preincrement
print $sum++; # postincrement
print ++$sum; # preincrement

print $sum--; # preincrement
print --$sum; # preincrement

# strings
$s = "Top 5";
$n = 10.0;
$add = $n + $s; # the string is evaluated in a numeric context, so Perl scans the string
print "Adding a number and a string:    $add\n";

$concat = $n . $s; # allows perl to find $n in the string
print "Concatenating a number and a string:     $concat\n";

$add2 = $concat + $add;
print "Adding the previous 2 results:   $add2\n\n";
```

The array example below is a snippet from the marble clock program. It showcases the dynamically sized array data type with push, pop, and shift functions.



```perl
#!/usr/bin/perl
use warnings;
$marbles = 0;
while($marbles<20)
{
    print "Number of marbles(>=20): "; #get input from user.
    $marbles = <>;
}
@reservoir = ();
for(1..$marbles) #enqueue reservoir.
{
    push(@reservoir, $_)
}
for(1..$marbles)
{
    push(@arr, $_)
}
$inOrder = '0';
$cycle=0;
while(not $inOrder)
{
    push(@oneMin, shift(@reservoir));
    while(scalar @reservoir != $marbles)
    {
        if(@oneMin != 4)
        {
            push(@oneMin, shift(@reservoir));
        }

        else
        {
            for(1..4)
            {
                push(@reservoir, pop(@oneMin));
            }

            if(@fiveMins !=2)
            {
                push(@fiveMins, shift(@reservoir));
            }

            else
            {
                for(1..2)
                {
                    push(@reservoir, pop(@fiveMins));
                }

                if(@fifteenMins !=3)
                {
                    push(@fifteenMins, shift(@reservoir));
                }
```

Below is an example of a simple hash that utilizes the pop function.



```
# hash
%hash = ( width => '400',
            height => '200',
                color => 'blue',
                    length => '100' );

print "\$hash { 'width' } = $hash{ 'width' }\n";
print "\$hash { 'height' } = $hash{ 'height' }\n";
print "\$hash { 'color' } = $hash{ 'color' }\n";
print "\$hash { 'length' } = $hash{ 'length' }\n";

# functions of hash: keys, values, each and reverse
@keys = keys( %hash );

while ( $key = pop( @keys ) ){ # function key returns a list of all the keys in the hash
    print "$key => $hash{ $key }\n";
}

@values = values( %hash );
print "\nThe values of the hash are:\n@values\n\n"; # will output: 400 200 blue 100

# reverse the hash and use function each to get each pair
print "%hash with its keys and values reversed\n";
%hash1 = reverse( %hash );
#print "%hash1";

while ( ( $key, $value ) = each( %hash ) ) { # function each keeps track of its location in the hash
    print "$key => $value\n";
}
```

The basic control abstractions of the language (loops, conditional controls, etc.)

In Perl, there are 11 available control structures:

1. sequence
2. if
3. unless
4. if/else
5. if/elsif/else
6. while
7. until
8. do/while
9. do/until
10. for
11. foreach

These control structures can be combined in only two ways, control-structure stacking, and control-structure nesting. Every Perl control structure must use {} to enclose the body of the loop. To manipulate control structures, Perl three loop keywords.

1. "next", to jump directly to the end of the current iteration.
2. "last", to terminate execution and ignore the "continue" block.
3. "redo" to restart the current iteration and ignore both the loop condition and continue block.

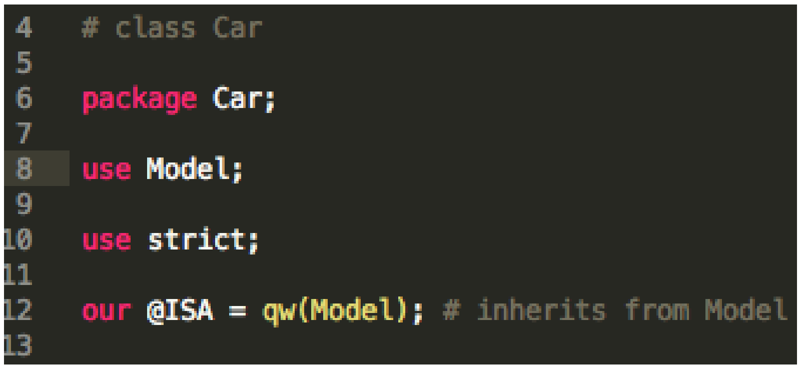
Control structures in Perl are similar to control structures in C and Java. If-statements run if the expression is true, and unless-statements run if the expression is false—similar to "if(!true)". Do/until statements run until a condition is met. The other control structures in Perl perform similarly to the control structures of C and Java.

There are conditional controls in Perl that dictate whether a control structure executes. The keywords “eq” and “ne” mean “equal to” and “not equal”, respectively. The “and” operator is identical to Java’s “&&”. It is also the same for the “or” operator, “||”. When Larry Wall was deciding which operators to make available to Perl, he didn’t want former C programmers to miss something that C had and Perl didn’t, so he brought over all of C’s operators to Perl.

How the language handles abstraction (including functions, procedures, objects, modules, etc.)

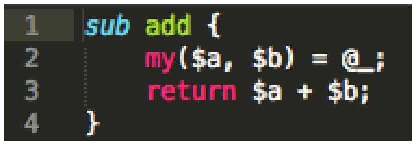
Abstraction in Perl is similar to other object-oriented languages. Elements from object-oriented languages like classes, modules, packages, methods, and attributes are all present in Perl. In order to create a class, a package must first be built. A class can be created with only one line of code, i.e. “package Car;”. Once the class is created, methods can be contained within the package. When an object method is called, it references the object passed along with other arguments. Concerning the dynamic scope nature of Perl, the package definition extends to the end of the file or until the interpreter reaches another package keyword.

Perl handles inheritance in a simple and easy manner. It has a special variable “@ISA” that handles inheritance. The code below shows a simple inheritance in Perl:



```
# class Car

package Car;

use Model;

use strict;

our @ISA = qw(Model); # inherits from Model

```

For the explanation of Perl procedures, it is better to analyze a basic example:



```
sub add {
    my($a, $b) = @_;
    return $a + $b;
}
```

The “add” method receives two parameters and returns the sum. The programmer could call “add” by writing “$z = add($x, $y);”. Formally, there is no parameter list. All parameters are passed in what is called a flat list, “@_” within the procedure. The parameters, $x and $y are passed by “call-by-reference”. If one of the parameters isn’t an l-value, the caller creates a reference copy of it implicitly. By assigning @_ to the local variables Perl achieves call-by-value semantics.

An evaluation of the language's writability, readability, and reliability

- **Readability**

Perl is very flexible. But it is so flexible sometimes that its ‘excessive’ redundant grammars may confuse Perl programmers, let alone those novices. Hence it has a ‘reputation’: write-only programming language. Which means it is a language with syntax (or semantics) sufficiently dense and bizarre that any routine of significant size is too difficult to understand by other non-Perl programmers and cannot be safely edited and maintained. Beginners are likely to have difficulties understanding, so they have do a lot of practice to make their codes elegant, and easy to be maintained. Hence Perl has a poor readability.

Even the father of Perl, Larry Wall, agrees to the poor readability of Perl code: “*Though I’ll admit readability suffers slightly…*”

- **Writability**

Since Perl is write-only, many features that hinder readability increases writability in Perl. As a good example, assigning the contents of a text file to a scalar variable can be realized in one line of code: {local $/; $content = <HANDLE>}. But non-Perl programmers would really find it difficult to understand what it is about (not readable), but it is sheer evidence of its powerful writability and robustness.

Perl provides us with some abstract data types such as arrays and hashes, which would otherwise have to be defined by users in most of programming languages. The incorporation of the data types is easy and thus increases the writability. The use of sigil to differentiate between data types is also a good example. Once you understand the all the features you can express your computation clearly, concisely and quickly.

Perl is very flexible and powerful. Perl Poetry is the practice of writing poems that can be compiled as legal Perl code, which helps programmers to have a deeper understanding of large number of English key words used in Perl. *Black Perl*, written by the founder of Perl Larry Wall, is very famous throughout Perl community (however it has not been compiled for years since it is written in Perl 3 in 1990 and will not parse in Perl 5). You can find numbers of poems at PerlMonks, which is a community website for Perl programmers.

- **Reliability**

With the development of Perl from 1987 to present, it has become more stable in terms of its reliability. According to the commentaries, it is as good as some closed source software or even better.

Perl has been labeled as a “duct tape of the Internet” since 1998. Such honor implies that Perl should be powerful, reliable and performs to its specifications under most conditions. A lot of people are using Perl to embed into their web pages to get things done more efficiently.
Reliability can be viewed as an efficiency issue. Perl is no guarantee of bug-free, we know that Perl has too many excessive redundant grammars and any mistyping may cause problems and there might not be any report of error. The core of Perl is: *There’s more than one way to do it*, but different programmers have different programming styles, it makes it hard for programmers to maintain others’ program. This is the problem of Perl’s reliability, but it doesn’t mean that it is not reliable.

The major strengths and weaknesses of the language.

Strengths:

- One of the biggest strength of the perl is text processing and especially Regular expression support. It's regular expression support is the most versatile in existence and seamlessly integrated into the language. Difficult programs like search and replace or statistical transformation or Pattern matching, can be done easily with the perl. It also provides functionality of Unix, awk, sed, tr etc. to make it more easier.
- The second major strength of perl is, it's large 3rd party modules i.e. CPAN (the Comprehensive Perl Archive Network) which is gateway to all things in perl. It has 127,209 perl modules in 28,657 distributions , written by 11,049 authors, mirrored on 273 servers. The archive has been online since 1995 and is still constantly growing. This huge collection of perl code is useful for so many purposes : Mathematics, Biology, Database access, Science, System administration, Foreign languages and Networking, etc.
- The third major strength of perl is Portability. Code of perl does not use system specific feature and because of it, Perl can be run on any platform. For example, if you write an utility that renames files in a complete directory tree according to some rule, you will be able to run it on Windows and Unix (Linux, Solaris, AIX, BSD, etc), and probably Mac OS too. But if you write an utility that changes file permissions, you might run into problems, because file permission are implemented differently on different systems.
- Programs can be mixable in the perl. The programs written in C/C++ can call Perl scripts to do different functions which are not supported by C/C++. It can also link to the C/C++ libraries. Like C++ language, Perl also support object oriented programming. It also Includes features common to many programming languages that are Low level, access to powerful built-in and system functions.
- Perl is free and It is also common enough on most modern Linux, and Solaris installations. Other useful feature of perl are Report generation, Scripting, and Rapid prototyping.

Weaknesses :

- Readability is the biggest weakness of the perl. Perl has very ugly syntax. There are lots of operators and special syntaxes which makes perl syntax very short but very complex too. This means if you stuck with any syntax errors, it will take some time to sort out that error.
- Maintenance of systems which are implemented in the perl are difficult. You cannot easily understand someone else scripts and it takes more time to maintain the system.
- Moreover, if you write a script which uses modules from CPAN, and want to run it on another computer, you need to install all the modules on that other computer, which takes more time with every new module.

An overview of the programs that you included and a discussion of what language features they highlight and how the language made the programs easy/hard to implement.

Marble Clock

We take the advantages of the data structures (queue and stack) supplied by Perl. We use queue to store marbles in reservoir and four stacks for other four trays respectively. The one on top has four slots each of them stands for one minute; the second tray has two slots each of them stands for 5 minutes; the third one is fifteen minutes tray with 3 slots and each of them stands for 15 minutes and the forth tray has 11 slots and each of them stands for 1 hour. **Our program will output the number of cycles (12-hours for 1 cycle) when all the marbles are back in their original order after they return to the reservoir (at 12am/pm).**

The **algorithm**:

let counter = 0;
1. **If** oneMin tray is not full then load the tray with one marble from reservoir and **go to step 1**. **Otherwise**, **go to step 2**.

2. Empty the oneMin tray and let all the marbles back into the reservoir from oneMin tray. **If** fiveMins tray is not full then load the tray with one marble from the reservoir and **go to step 1. Otherwise**, **go to step 3**.

3. Empty the fiveMins tray and let all the marbles back into the reservoir from fiveMins tray. **If** fifteenMins tray is not full then load the tray with one marble from the reservoir and **got to step 1**. **Otherwise**, **go to step 4**.

4. Empty the fifteenMins tray and let all the marbles back into the reservoir from fifteenMins tray. **If** hours tray is not full then load the tray with one marble from the reservoir and **go to step 1**. **Otherwise**, **go to step 5**.

5. Empty the hours tray and let all the marbles back into the reservoir from hours tray. **Increase counter by one**. Check **if** the marbles are in original orders; if not, **go to step 1**. **Otherwise**, return the value of counter and **done**.

We use array stacks for non-reservoir trays and a queue for reservoir, since their features are the same as the descriptions of the marble clock. Perl has several functions for array, which makes it easy to realize all the functions of array stack and array queue, so we don't need to create array stack and array class.

The push() function push an element to the end of the array which is used for both stack and queue. The pop() function remove and return the last element, which reduces the elements by 1 which is for stack. The shift() function remove and return the first element from the array and reduce the size of the array by 1, which is used for queue (first in first out).
Since the array in Perl is flexible, we don't need declare the length of the array, the size of the array will change with the number of elements it contains. Furthermore, we don't need two variables (to record where front and rear are) for queue, since the array is flexible. And we don't need a variable for (to record the top) stack. If you want to peek (function in stack and queue) the stack of queue, just return

the value of last element. You don’t need a variable for the size of stack or queue; you can just check the size of the array. This is the advantage of dynamic programming language.

Three of A Crime

We have used some to the Perl data structures to complete this program, including arrays and a hash. We used arrays to hold All of the criminals, the actual criminals, the fake criminals, and the criminals to be showed to the players. The undef foreach hash function was used to create a hash to hold the actual criminals, using the hash made it easier to create the array for the fake criminals using grep(EXPR,LIST) function. The function evaluates EXPR for each element of LIST and returns the array value consisting of those elements for which the expression evaluated to true. In our case the EXPR not exist and the list was the elements in the hash which is the real criminals.

We have also used Perl rand() function which generate random numbers. The function was used along with while loops to make a three discrete random choices for the actual criminals, and to make a random selections from the actual criminals and fake criminals to be shown to the player as well as to randomizes the order of cards displayed to the player.

Finally we have used Perl TK module to create the user graphics interface. This module made it way easier to create the GUI. We have created diffident frames for different purposes, then using pack () and packForget() to switch between those frames. We have also used the configure function to edit the text of the label after it been created to show different set of cards each time.

Other Programs

We also implement other five programs which demonstrate following features of perl.

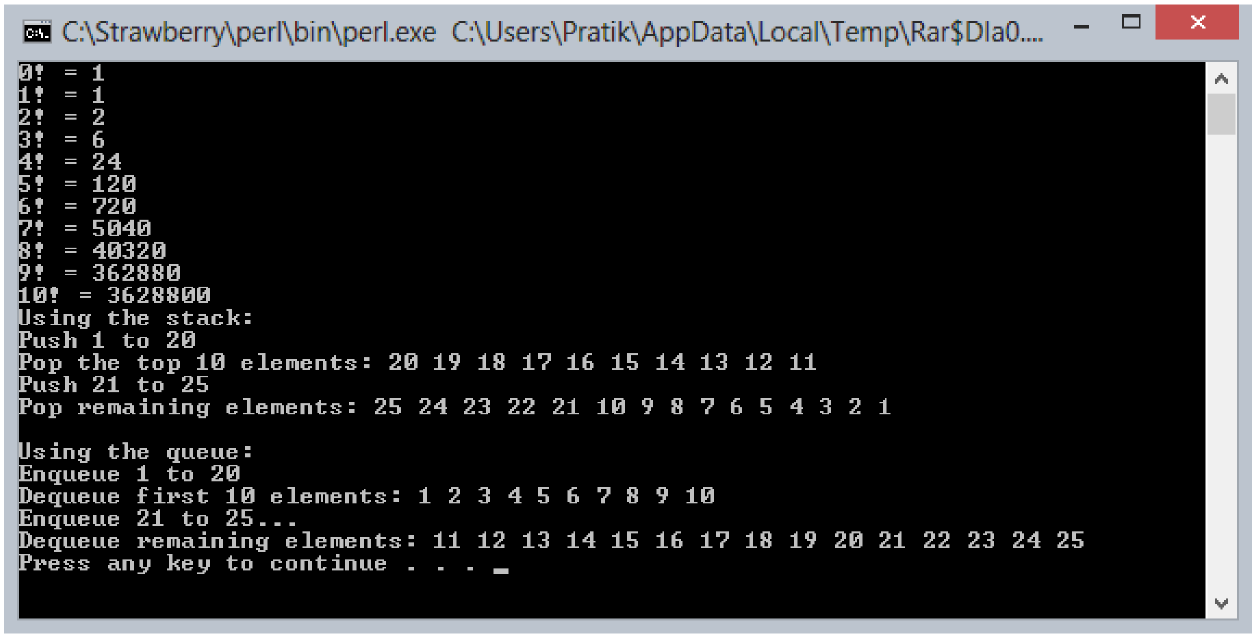
1. Unique and advanced features of Perl (advFeatures.pl)
2. All available data types in perl (AllavailableDataTypes.pl)
3. All available control structures in perl (ControlStructures.pl)
4. String Manipulation in perl (strMan.pl)

This is a program demonstrating unique advanced features of Perl

advFeatures.pl

```
#!/usr/bin/perl -
#shebang construct(#!)
# this is a program demonstrating unique advanced features of Perl
use warnings;

# recursive subroutine
foreach ( 0 .. 10 ) { # call function factorial for each of the numbers from 0 through 10 and display results
	print "$_! = " . factorial( $_ ) . "\n";
}

sub factorial
{
	my $number = shift; # get the argument

	if ( $number <= 1 ) { # base case
		return 1;
	}
	else { # recursive step
		return $number * factorial( $number - 1 );
	}
}

# implementing stacks and queues

my ( @stack, @queue );
print( "Using the stack:\nPush 1 to 20\n" );

for ( 1 .. 20 ) {
	push( @stack, $_ );
}

print( "Pop the top 10 elements: " );

for ( 1 .. 10 ) {
	print( pop( @stack ), " " );
}

print( "\nPush 21 to 25\n" );

for ( 21 .. 25 ) {
	push( @stack, $_ );
}

print( "Pop remaining elements: " );

while ( @stack ) {
	print( pop( @stack ), " " );
}

print( "\n\nUsing the queue:\nEnqueue 1 to 20\n" );

for ( 1 .. 20 ) {
	push( @queue, $_ );
}

print( "Dequeue first 10 elements: " );

for ( 1 .. 10 ) {
	print( shift( @queue ), " " );
}

print( "\nEnqueue 21 to 25...\n" );

for ( 21 .. 25 ) {
	push( @queue, $_ );
}

print( "Dequeue remaining elements: " );

while ( @queue ) {
	print( shift( @queue ), " " );
}

print "\n";
```

Output :


C:\Strawberry\perl\bin\perl.exe C:\Users\Pratik\AppData\Local\Temp\Rar$DIa0....

```
0! = 1
1! = 1
2! = 2
3! = 6
4! = 24
5! = 120
6! = 720
7! = 5040
8! = 40320
9! = 362880
10! = 3628800
Using the stack:
Push 1 to 20
Pop the top 10 elements: 20 19 18 17 16 15 14 13 12 11
Push 21 to 25
Pop remaining elements: 25 24 23 22 21 10 9 8 7 6 5 4 3 2 1

Using the queue:
Enqueue 1 to 20
Dequeue first 10 elements: 1 2 3 4 5 6 7 8 9 10
Enqueue 21 to 25...
Dequeue remaining elements: 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25
Press any key to continue . . .
```

This is a program demonstrating all available data types in perl

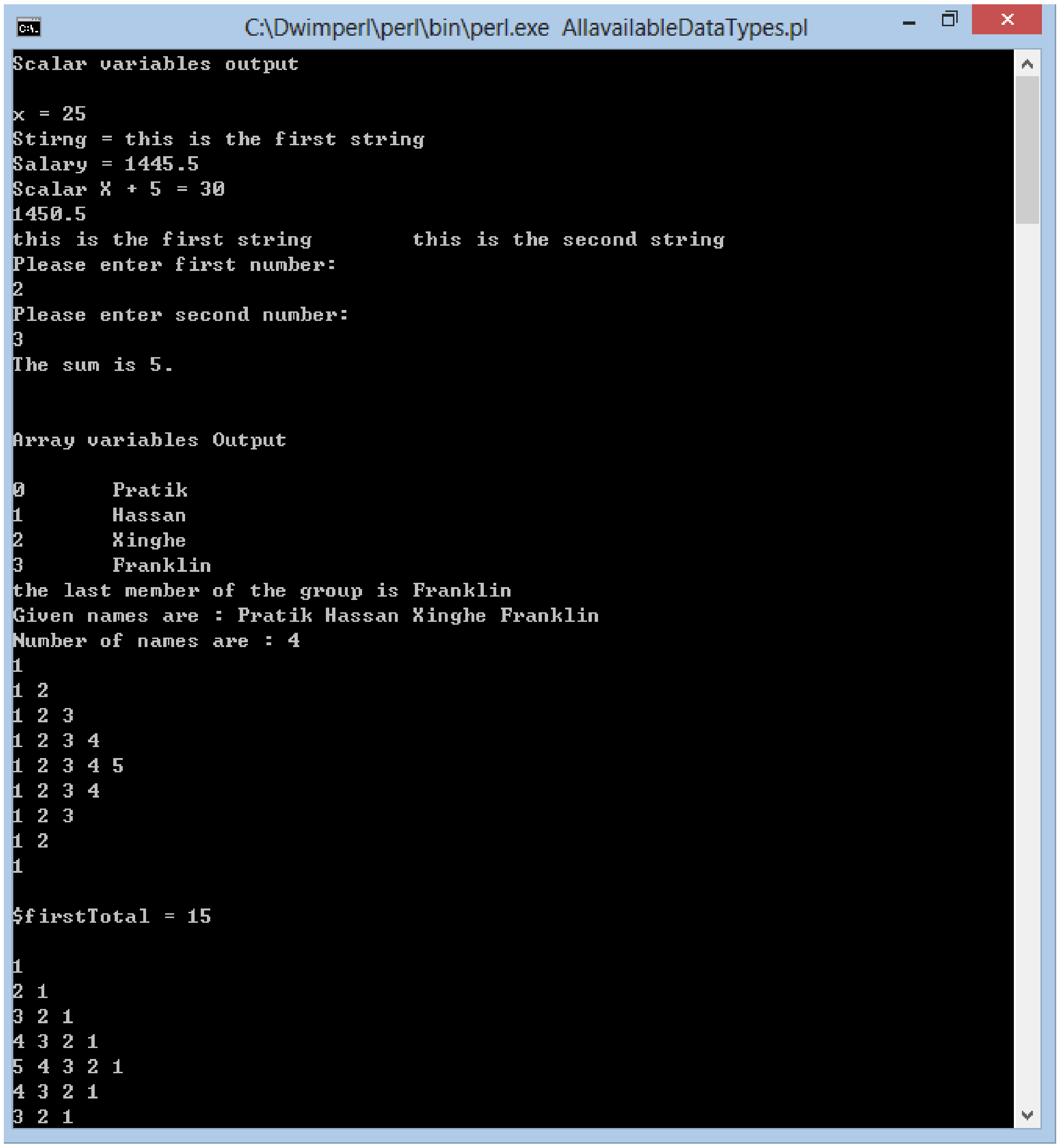
AllavailableDataTypes.pl

```
#!/usr/bin/perl -
# shebang construct(#!)
# this is a program demonstrating all available data types in perl
# there are 3 types of data types in perl: 1. scalars 2. arrays 3. hashes
use warnings;

# scalar variables -- dynamically typed and dynamic casted
# wherever a scalar is used it is always preceded by a dollar sign
#scalar
print "Scalar variables output \n\n";
$x = 25;                        # An integer assignment
$s = "this is the first string";        # A string
$s2 = "this is the second string";
$salary = 1445.50;                      # A floating point
print "x = $x \n";
print "Stirng = $s \n";
print "Salary = $salary \n";

print "Scalar X + 5 = ";
print $x + 5  ."\n";
print $salary + 5.0 ."\n";

$concat = $s ."\t". $s2;
print "$concat \n";

print "Please enter first number:\n";
$n1 = <STDIN>; # STDIN is an abbreviation for standard input
chomp $n1; # chomp the \n

print "Please enter second number:\n";
$n2 = <STDIN>;
chomp $n2;

$sum = $n1 + $n2; # there are different types of operators like +, -, *, /, %, **

print "The sum is $sum.\n";

# arrays
print "\n\nArray variables Output \n\n";
@arrayNames = ("Pratik", "Hassan", "Xinghe", "Franklin");

$i = 0;
# print our names
```

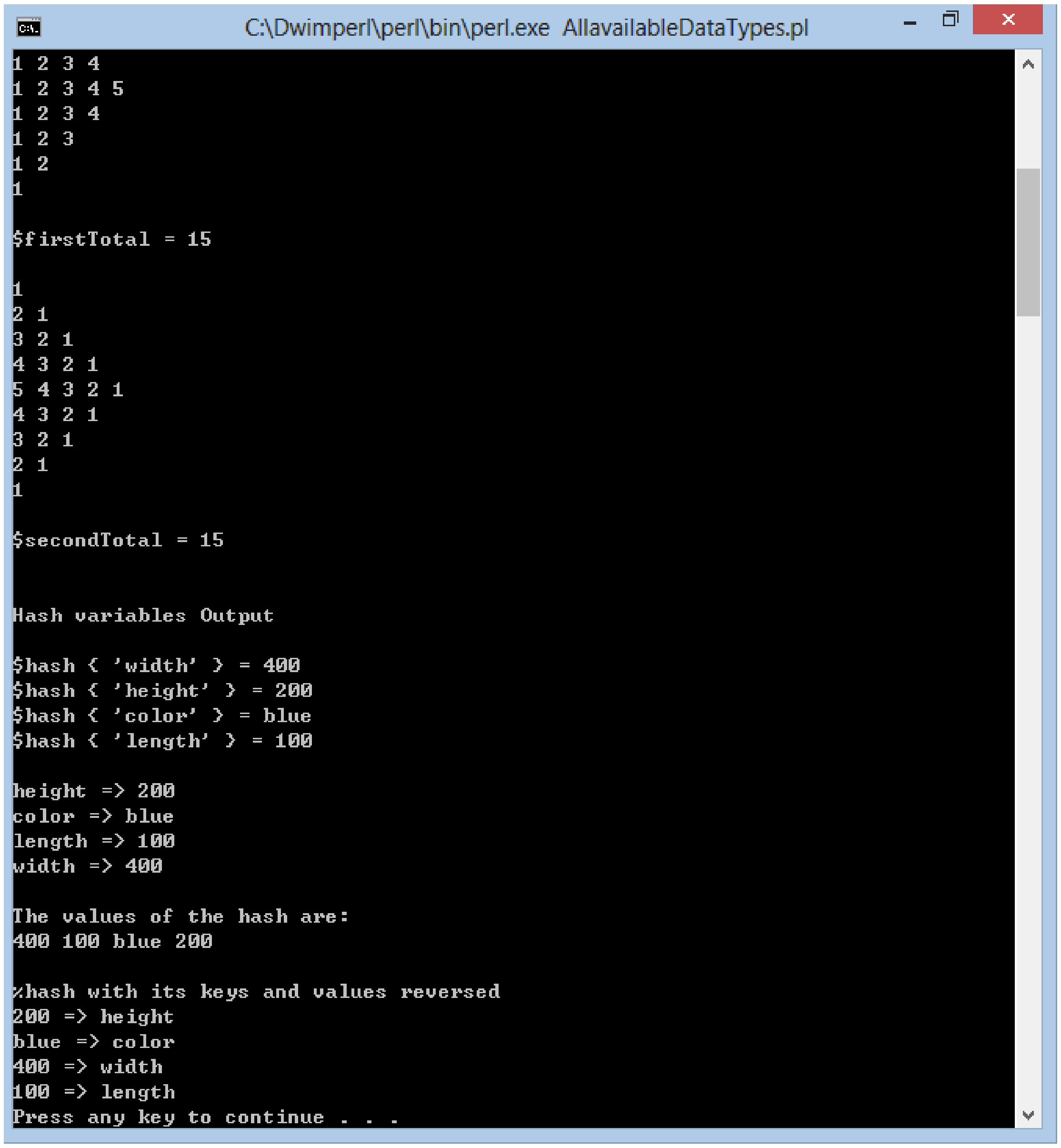
```
while ( $i < 4 ) {
            print "$i            $arrayNames[ $i ]\n";
            ++$i;
}

print "the last member of the group is $arrayNames[3]\n";

@copy = @arrayNames;
$size = @arrayNames;

print "Given names are : @copy\n";
print "Number of names are : $size\n";

# push, pop, shift, unshift functions of the arrays
# use push to insert elements at the end of the array
for ( $i = 1; $i <= 5; ++$i ) {
            push (@array, $i);
            print "@array\n";
}

# use pop to remove elements
while ( @array ) {
            $firstTotal += pop( @array );
            print "@array\n";
}

print "\$firstTotal = $firstTotal\n\n";

# use unshift to insert elements at the front of @array
for ( $i = 1; $i <= 5; ++$i ) {
            unshift( @array, $i );
            print "@array\n";
}

# use shift to remove excess elements
while ( @array ) {
            $secondTotal += shift( @array );
            print "@array\n";
}

print "\$secondTotal = $secondTotal\n";

# hash
print "\n\nHash variables Output \n\n";
%hash = ( width => '400',
                        height => '200',
                                    color => 'blue',
                                                length => '100' );
```

```
print "\$hash { 'width' } = $hash{ 'width' }\n";
print "\$hash { 'height' } = $hash{ 'height' }\n";
print "\$hash { 'color' } = $hash{ 'color' }\n";
print "\$hash { 'length' } = $hash{ 'length' }\n\n";

# functions of hash: keys, values, each and reverse
@keys = keys( %hash );

while ( $key = pop( @keys ) ){ # function key returns a list of all the keys in the hash
	print "$key => $hash{ $key }\n";
}

@values = values( %hash );
print "\nThe values of the hash are:\n@values\n\n"; # will output: 400 200 blue 100

# reverse the hash and use function each to get each pair
print "%hash with its keys and values reversed\n";
%hash = reverse( %hash );

while ( ( $key, $value ) = each( %hash ) ) { # function each keeps track of its location in the hash, so every
call to this function returns a new key-value pair
	print "$key => $value\n";
}
```

Output :


C:\Dwimperl\perl\bin\perl.exe AllavailableDataTypes.pl

```
Scalar variables output

x = 25
Stirng = this is the first string
Salary = 1445.5
Scalar X + 5 = 30
1450.5
this is the first string        this is the second string
Please enter first number:
2
Please enter second number:
3
The sum is 5.


Array variables Output

0       Pratik
1       Hassan
2       Xinghe
3       Franklin
the last member of the group is Franklin
Given names are : Pratik Hassan Xinghe Franklin
Number of names are : 4
1
1 2
1 2 3
1 2 3 4
1 2 3 4 5
1 2 3 4
1 2 3
1 2
1

$firstTotal = 15

1
2 1
3 2 1
4 3 2 1
5 4 3 2 1
4 3 2 1
3 2 1
```


C:\Dwimperl\perl\bin\perl.exe AllavailableDataTypes.pl

```
1 2 3 4
1 2 3 4 5
1 2 3 4
1 2 3
1 2
1

$firstTotal = 15

1
2 1
3 2 1
4 3 2 1
5 4 3 2 1
4 3 2 1
3 2 1
2 1
1

$secondTotal = 15


Hash variables Output

$hash { 'width' } = 400
$hash { 'height' } = 200
$hash { 'color' } = blue
$hash { 'length' } = 100

height => 200
color => blue
length => 100
width => 400

The values of the hash are:
400 100 blue 200

%hash with its keys and values reversed
200 => height
blue => color
400 => width
100 => length
Press any key to continue . . .
```

This is a program demonstrating all available control structures in perl

ControlStructures.pl

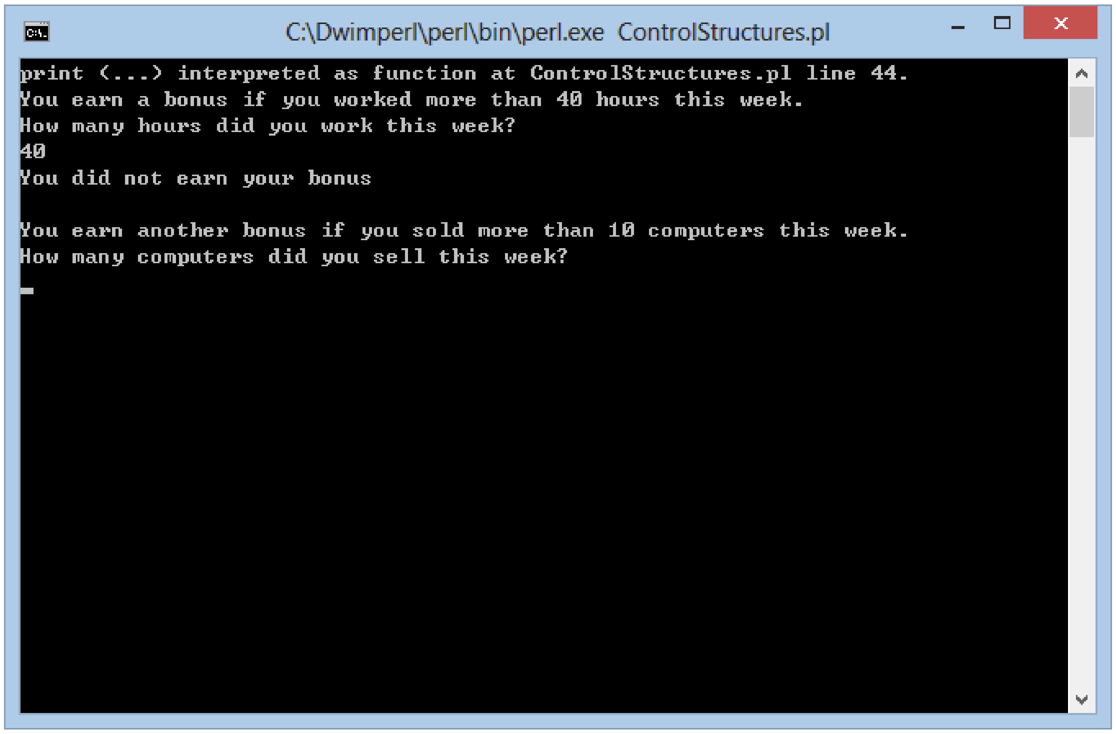
```
#!/usr/bin/perl -
#shebang construct(#!)
# this is a program demonstrating all available control structures in perl
# there are 11 available control stuctures: 1. sequence 2. if 3. unless 4. if/else 5. if/elsif/else 6. while 7.
until 8. do/while 9.  do/until 10. for 11. foreach
# these control structures can be combined in only two ways: 1. control-structure stacking 2. control-
structure nesting
# every perl control structure must use {} to enclose the body of the loop
use warnings;

# if and unless Selection structures
print "You earn a bonus if you worked more than 40 hours this week.\n";
print "How many hours did you work this week?\n";
$hours = <STDIN>; # get input from user

# if statement occurs if condition is ture
if ( $hours > 40 ) {
	print "You earned your bonus!\n";
}

# if only one task is to be completed when a condition is met, such as in a print statement, perl allows to
be written like this:
print "You work too much!\n" if $hours >= 100; # this if statement is identical to the one above

# unless statement occurs if condition is false
unless ( $hours > 40 ) {
	print "You did not earn your bonus\n";
}

# can also be written in one line like the if statement above
print "You work too little!\n" unless $hours >= 20;

# if/else statement
print "\nYou earn another bonus if you sold more than 10 computers this week.\n";
print "How many computers did you sell this week?\n";
$sales = <>;

if ( $sales > 10 ) {
	print "You earned the sales bonus!\n";
}
else {
	print "You did not earn the sales bonus\n";
}
```

```
# conditional operator (?:), Perl's only ternary operator; it takes three operands, The operands together
with the conditional operator, form a conditional expression.
# the first operand is a condition, the second operand is the value for the entire conditional expression if
the condition is true, and the third operand is the value for the entire conditional expression if the
condition is false
print ( $sales > 0 ? "You managed to sell some computers, you will still have a job next week\n" :
                                        "You didn't sell any computers, you are fired\n" );

# ^ can also be rewritten like this:
$sales >= 25 ? print "Wow you did great this week! You will be promoted soon!\n" :
                                print "Keep up the good work!\n";

# if/elsif/else statement tests multiple cases
if ( $sales >= 25 ) {
        print "\$2,000 bonus!\n";
}
elsif ( $sales >= 15 ) {
        print "\$1,000 bonus!\n";
}
elsif ( $sales > 10 ) {
        print "\$500 bonus!\n";
}
else {
        print "Maybe next week you can get a bonus\n";
}

# while loop repeats while its condition is true
$num = 0;
while ( $num <= 3 ) {
        $num += 1; # increments by one until 3
}

# until loop is the opposite of while. Repeats while its condition is false
until ( $num > 3 ) {
        $num += 1; # increments by one until 3
}

# do/while & do/until tests loop-continuation after the body of the loop is performed. The body of the
loop is performed at least once.
print "\ndo while example:\n";
$count = 1;
do {
        print "$count ";
} while ( ++$count <= 5 ); # do/while

print "\ndo until example:\n";
```

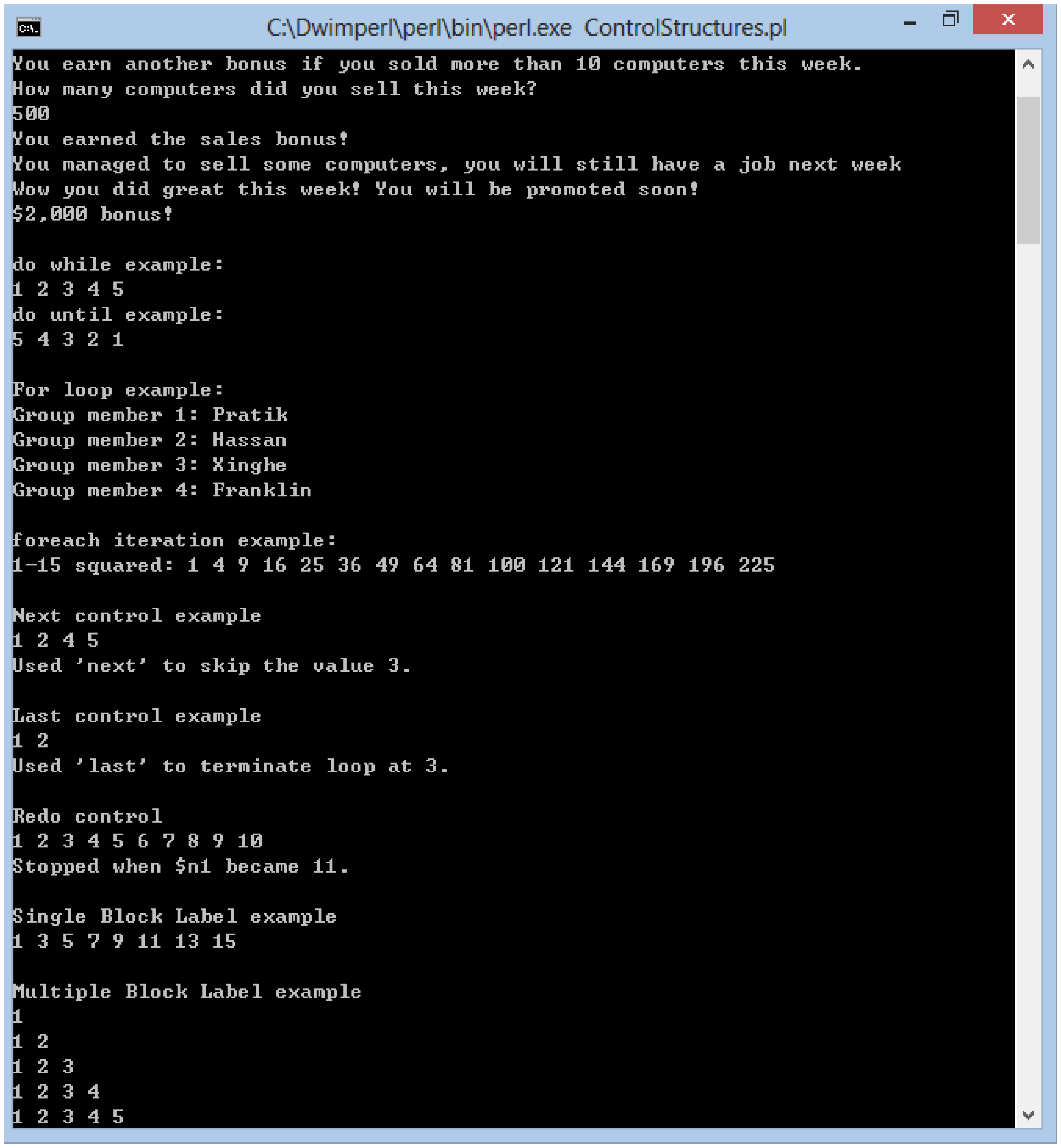
```
$count = 5;
do {
	print "$count ";
} until ( --$count == 0 ); # do/until

print "\n\nFor loop example:\n";

# for repetition structure
# more appropriate when looping through an array of elements
@array = ("", "Pratik", "Hassan", "Xinghe", "Franklin");

for ( $i = 1; $i < 5; ++$i ) {
	print "Group member $i\: $array[ $i ]\n";
}

print "\nforeach iteration example:\n";

# foreach repetition structure allows a programmer to iterate over a list of values, accessing and
manipulating each element.
@set = ( 1 .. 15 );

foreach $number ( @set ) {
	$number **= 2;
}

print "1-15 squared: @set\n";

# loop controls: next
print "\nNext control example\n";

@list = ( 1 .. 5 );
foreach $x ( @list ) {

	if ( $x == 3 ) {
		$skipped = $x; # store 3, the skipped value
		next; # skip remaining code in loop only if $_ is 3
	}
	else {
		print "$x ";
	}
}

print "\nUsed 'next' to skip the value $skipped.\n\n";

# loop controls: last
print "Last control example\n";
foreach $x ( @list ) {
		if ( $x == 3 ) {
			$number = $x; # store the current value before loop ends
			last; # jump to end of foreach structure
		}
		else {
			print "$x ";
		}
}

print "\nUsed 'last' to terminate loop at $number.\n\n";

# loop controls: redo
print "Redo control\n";
$n1 = 1;

while ( $n1 <= 5 ) {

		if ( $n1 <= 10 ) {
			print "$n1 ";
			++$n1;
			redo; # continue loop without testing ( $n1 <= 5 )
		}
}

print "\nStopped when \$n1 became $n1.\n\n";

print "Single Block Label example\n";
# single block label - describes a block of code and acts as a target for the loop control commands next,
last, and redo.
LOOP: for ( $number = 1; $number <= 15; ++$number ) {
		next LOOP if ( $number % 2 == 0 );
		print "$number "; # displays only odd numbers from 1 - 15
}

print "\n\nMultiple Block Label example\n";
# multiple block label
OUTER: foreach $row ( 1 .. 10 ) {

		INNER: foreach $column ( 1 .. 10 ) {

			if ( $row < $column ) {
				print "\n";
				next OUTER;
			}

			print "$column ";
		}
}
```

```
print "\n\nBare Block example\n";
# bare blocks - a block of code enclosed by curly braces, with or without a label, but with no
accompanying control-structure keyword. 'next' is equivalent to 'last' in a bare-block context
print "What number am I thinking of? Guess a number between 1 and 3: ";
$guess = <STDIN>;

BLOCK: { # start of bare block
	if ( $guess == 1 ) {
		print "You guessed it!\n";
		last BLOCK; # jump to the end of the block
	}
	if ( $guess == 2 ) {
		print "You are close!\n";
		last BLOCK;
	}
	if ( $guess == 3 ) {
		print "Nope, that's not right!\n";
		last BLOCK;
	}
	#default case if $guess isn't 1, 2, or 3
	{
		print "Guess again a new number between 1 and 3!: ";
		$guess = <STDIN>;
		redo BLOCK;
	}
}

# Error functions: die - terminate program execution and prints a message
print "\nChoose a positive number (input a negative number to show the 'die' error message): ";
$z = <STDIN>;
unless ( $z > 0 ) {
	die "Error: You didn't read my instructions!\nThis program will now terminate!\nGoodbye\n";
}

#Error functions: warn - produces same output as 'die' but program execution continues
print "Choose a negative number (input a positive number to show the 'warn' error message): ";
$a = <STDIN>;
unless ( $a < 0 ) {
	warn("Error: You didn't read my instructions!\nThis program will still continue\n");
}

print "\n\nEnd of control structures program\n";
```

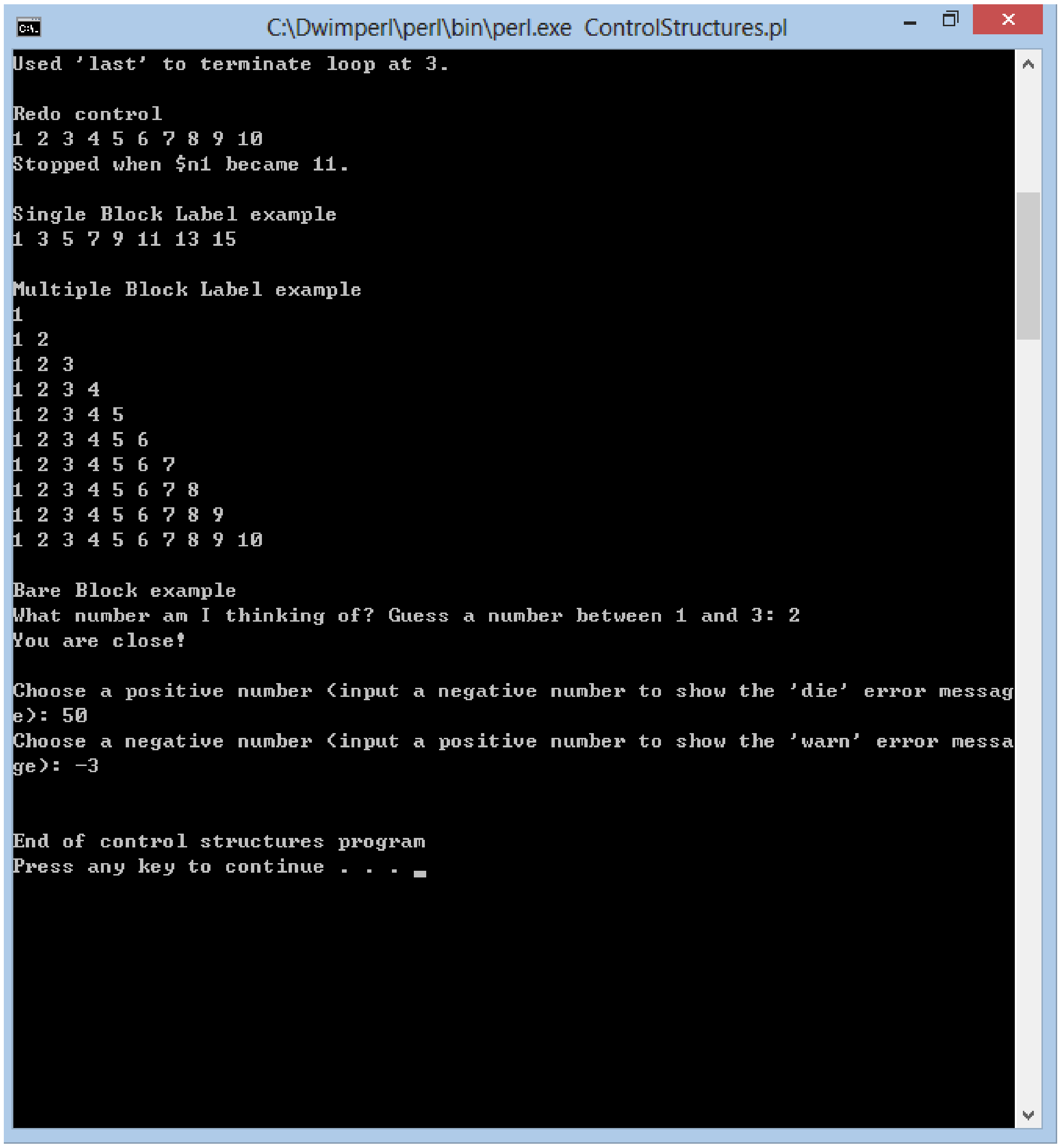
Output :



```
print (...) interpreted as function at ControlStructures.pl line 44.
You earn a bonus if you worked more than 40 hours this week.
How many hours did you work this week?
40
You did not earn your bonus

You earn another bonus if you sold more than 10 computers this week.
How many computers did you sell this week?
```


C:\Dwimperl\perl\bin\perl.exe ControlStructures.pl

```
You earn another bonus if you sold more than 10 computers this week.
How many computers did you sell this week?
500
You earned the sales bonus!
You managed to sell some computers, you will still have a job next week
Wow you did great this week! You will be promoted soon!
$2,000 bonus!

do while example:
1 2 3 4 5
do until example:
5 4 3 2 1

For loop example:
Group member 1: Pratik
Group member 2: Hassan
Group member 3: Xinghe
Group member 4: Franklin

foreach iteration example:
1-15 squared: 1 4 9 16 25 36 49 64 81 100 121 144 169 196 225

Next control example
1 2 4 5
Used 'next' to skip the value 3.

Last control example
1 2
Used 'last' to terminate loop at 3.

Redo control
1 2 3 4 5 6 7 8 9 10
Stopped when $n1 became 11.

Single Block Label example
1 3 5 7 9 11 13 15

Multiple Block Label example
1
1 2
1 2 3
1 2 3 4
1 2 3 4 5
```



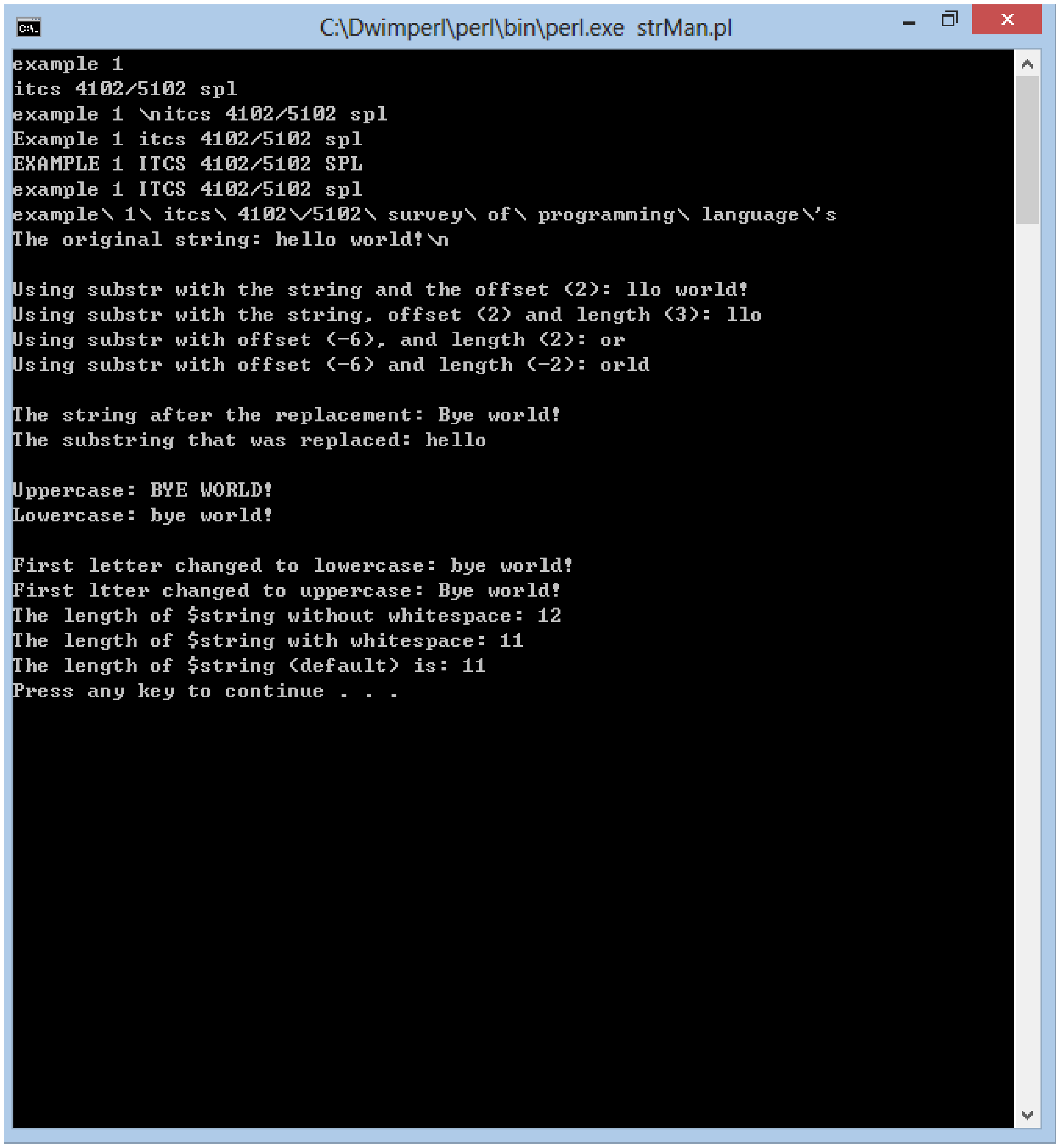
```
Used 'last' to terminate loop at 3.

Redo control
1 2 3 4 5 6 7 8 9 10
Stopped when $n1 became 11.

Single Block Label example
1 3 5 7 9 11 13 15

Multiple Block Label example
1
1 2
1 2 3
1 2 3 4
1 2 3 4 5
1 2 3 4 5 6
1 2 3 4 5 6 7
1 2 3 4 5 6 7 8
1 2 3 4 5 6 7 8 9
1 2 3 4 5 6 7 8 9 10

Bare Block example
What number am I thinking of? Guess a number between 1 and 3: 2
You are close!

Choose a positive number (input a negative number to show the 'die' error messag
e): 50
Choose a negative number (input a positive number to show the 'warn' error messa
ge): -3


End of control structures program
Press any key to continue . . .
```

This is a program demonstrating String Manipulation in perl

StrMan.pl

```
#!/usr/bin/perl -
#shebang construct(#!)
use warnings;
use strict;

# This is case of interpolation.
my $str1 = "example 1 \nitcs 4102/5102 spl";
print "$str1\n";

# This is case of non-interpolation.
$str1 = 'example 1 \nitcs 4102/5102 spl';
print "$str1\n";

# Only e will become upper case.
$str1 = "\uexample 1 itcs 4102/5102 spl";
print "$str1\n";

# \U: Whole line will become capital.
$str1 = "\Uexample 1 itcs 4102/5102 spl";
print "$str1\n";

# \U...\EA portion of line will become capital.
$str1 = "example 1 \Uitcs 4102/5102 \Espl";
print "$str1\n";

# Backsalash non alpha-numeric including spaces.
$str1 = "\Qexample 1 itcs 4102/5102 survey of programming language's";
print "$str1\n";

my $string = "hello world!\n";
print "The original string: ", 'hello world!\n', "\n\n";

# Using substr
print "Using substr with the string and the offset (2): ";
print substr( $string, 2 );
print "Using substr with the string, offset (2) and length (3): ";
print substr( $string, 2, 3 ), "\n";
print "Using substr with offset (-6), and length (2): ";
print substr( $string, -6, 2 ), "\n";
print "Using substr with offset (-6) and length (-2): ";
print substr( $string, -6, -2 ), "\n\n";

# replace first 5 characters of $string with "bye"
# assign substring that was replaced to $substring
my $substring = substr( $string, 0, 5, "Bye" );

print "The string after the replacement: $string";
print "The substring that was replaced: $substring\n\n";

# convert all letters of $string to uppercase
$string = uc( $string );
print "Uppercase: $string";

# convert all letters of $string to lowercase
$string = lc( $string );
print "Lowercase: $string \n";

# only change first letter to lowercase
$string = lcfirst( $string );
print "First letter changed to lowercase: $string";

# only change first letter to uppercase
$string = ucfirst( $string );
print "First ltter changed to uppercase: $string";

# calculating the length of $string
my $length = length( 'Bye world!\n' );
print "The length of \$string without whitespace: $length \n";

$length = length( "Bye world!\n" );
print "The length of \$string with whitespace: $length \n";

$length = length( $string );
print "The length of \$string (default) is: $length \n";
```

Output :



C:\Dwimperl\perl\bin\perl.exe strMan.pl

```
example 1
itcs 4102/5102 spl
example 1 \nitcs 4102/5102 spl
Example 1 itcs 4102/5102 spl
EXAMPLE 1 ITCS 4102/5102 SPL
example 1 ITCS 4102/5102 spl
example\ 1\ itcs\ 4102\/5102\ survey\ of\ programming\ language\'s
The original string: hello world!\n

Using substr with the string and the offset (2): llo world!
Using substr with the string, offset (2) and length (3): llo
Using substr with offset (-6), and length (2): or
Using substr with offset (-6) and length (-2): orld

The string after the replacement: Bye world!
The substring that was replaced: hello

Uppercase: BYE WORLD!
Lowercase: bye world!

First letter changed to lowercase: bye world!
First ltter changed to uppercase: Bye world!
The length of $string without whitespace: 12
The length of $string with whitespace: 11
The length of $string (default) is: 11
Press any key to continue . . .
```

Common Programs

Three of a Crime

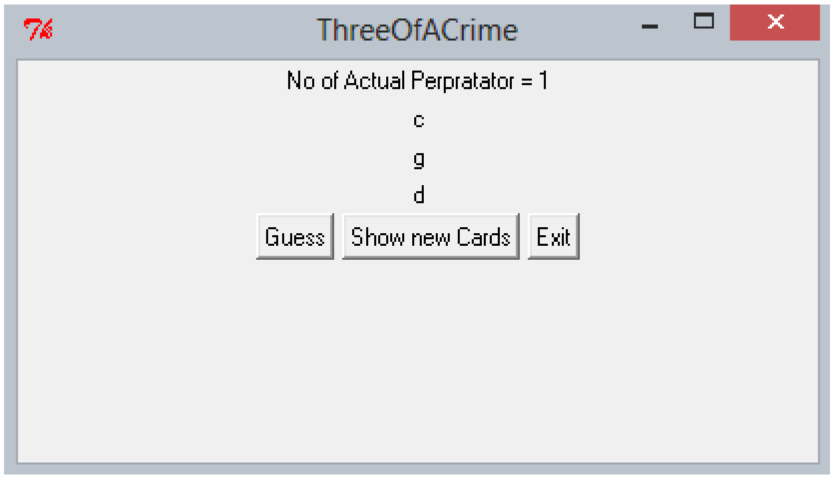
```
#!/usr/local/bin/perl
## This program demonstrates the Three of A Crime Game in Perl
use Tk;

@criminalNames = ('a', 'b', 'c', 'd','e', 'f', 'g'); # Array of the criminals

$a = int(rand(7)); #Select the random numbers
$b = int(rand(7));
$c = int(rand(7));


while( $b == $a)
{
	$b = int(rand(7));
}
while( $c == $a || $c==$b)
{
	$c = int(rand(7));
}

@ap =($criminalNames[$a],$criminalNames[$b],$criminalNames[$c]); # Array of the actual perpetrators


#---------------------------------------------------------------------------------------------------------------------
#-----------------------------------------------------     GUI     ------------------------------------------------------
#---------------------------------------------------------------------------------------------------------------------


#+++++++++++++++++++++++++++++++++++++++++++++++++++++++++  Cards Frame
+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
my $mw = new MainWindow; # Main Window
$mw->geometry("400x200");

my $frm_cards = $mw -> Frame() -> pack(); #New Frame
my $labNum = $frm_cards -> Label()->pack;
my $lab1 = $frm_cards -> Label()->pack;
my $lab2 = $frm_cards -> Label()->pack;
my $lab3 = $frm_cards -> Label()->pack;
&ShowCard();

my $but_guess = $frm_cards -> Button(-text=>"Guess", -command =>\&Guess) -> pack(-side=>left, -
padx => 2);
```

```
my $but_showcards = $frm_cards -> Button(-text=>"Show new Cards", -command =>\&ShowCard) ->
pack(-side=>left, -padx => 2);
my $but_exit = $frm_cards -> Button(-text => "Exit",-command => sub { exit })-> pack(-side=>left, -padx
=> 2);
#============================================================ end of Cards Frame
=======================================================


#+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++  Guess Frame
+++++++++++++++++++++++++++++++++++++++++++++++++++++++++
my $frm_guess  = $mw -> Frame(); #New Frame
my $label_erorr = $frm_guess -> Label();
my $label_firtGuess = $frm_guess -> Label(-text=>"Enter the first name of actual perpetrators :") ->
pack();
my $entry_firstGuess = $frm_guess -> Entry() -> pack();

my $label_secondGuess = $frm_guess -> Label(-text=>"Enter the second name of actual perpetrators :")
-> pack();
my $entry_secondGuess = $frm_guess -> Entry() -> pack();

my $label_thirdGuess = $frm_guess -> Label(-text=>"Enter the third name of actual perpetrators :") ->
pack();
my $entry_thirdGuess = $frm_guess -> Entry() -> pack();

my $but_enter = $frm_guess -> Button(-text=>"Enter", -command =>\&Result) -> pack(-side=>'left' ,  -fill
=> 'both',-expand => 1,-pady => 10,-padx => 2);
my $but_exit1 = $frm_guess -> Button(-text=>"Exit",-command => sub { exit })-> pack(-side=>'right',  -fill
=> 'both',-expand => 1,-pady => 10,-padx => 2);

#========================================================== end of Cards Frame
=======================================================


#+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++  Result Frame
+++++++++++++++++++++++++++++++++++++++++++++++++++++++++
my $frm_result  = $mw -> Frame(); #New Frame
my $label_resutl = $frm_result -> Label() -> pack();
my $but_exit2 = $frm_result -> Button(-text=>"Exit",-command => sub { exit })-> pack(-side=>'right',  -fill
=> 'both',-expand => 1,-pady => 10,-padx => 2);
#========================================================== end of Result Frame
=========================================================


#-----------------------------------------------------------------------------------------------------------------------------
#----------------------------------------------------    End of GUI   -----------------------------------------------------
#-----------------------------------------------------------------------------------------------------------------------------

MainLoop;
```

```
sub ShowCard
{
            my %hash;
            $hash{$_} = undef foreach (@ap);
            @criminalNames1 = grep { not exists $hash{$_} } @criminalNames; # Array of the fake
perpetrators



            $j = int(rand(3));

            if($j == 0)
            {
                        $actualPerprataro = "No of Actual Perpratator = 1";# Print the No of Actual perpetrators

                        $a = int(rand(4));
                        $b = int(rand(4));
                        while($a == $b)
                        {
                                    $a = int(rand(4));
                        }
                        @card = ($criminalNames1[$a],$criminalNames1[$b],$ap[int(rand(3))]);
            }


            if ($j == 1)
            {
                        $actualPerprataro = "No of Actual Perpratator = 2"; # Print the No of Actual perpetrators
                        $a = int(rand(3));
                        $b = int(rand(3));
                        while($a == $b)
                        {
                                    $a = int(rand(3));
                        }
                        @card = ($ap[$a],$ap[$b],$criminalNames1[int(rand(4))]);
            }

            if ($j == 2)
      {
              $actualPerprataro = "No of Actual Perpratator = 0"; # Print the No of Actual perpetrators
              $a = int(rand(4));
              $b = int(rand(4));
              $c = int(rand(4));

              while( $b == $a)
               {
                $b = int(rand(4));
            }

           while( $c == $a || $c==$b)
            {
                $c = int(rand(4));
            }

            @card = ($criminalNames1[$a],$criminalNames1[$b],$criminalNames1[$c]);
    }

        #Mix the selected cards.
        $a = int(rand(3));
        $b = int(rand(3));
        $c = int(rand(3));

        while( $b == $a)
        {
                $b = int(rand(3));
        }

        while( $c == $a || $c==$b)
        {
                $c = int(rand(3));
        }


        #show the new cards on the GUI
        $labNum -> configure(-text=>$actualPerprataro);
        $lab1 -> configure(-text=>$card[$a]);
        $lab2 -> configure(-text=>$card[$b]);
        $lab3 -> configure(-text=>$card[$c]);
}

sub Guess #change the fram from cards fram to guess fram.
{
        $frm_cards-> packForget();
        $frm_guess-> pack();
}

sub Result
{
        if ($entry_firstGuess -> get() eq "" || $entry_secondGuess-> get() eq "" || $entry_thirdGuess->
get() eq "" )
        {
                $label_erorr -> configure(-text=>"Please enter a value in all field", -foreground => red);
                $label_erorr -> pack();
        }
```

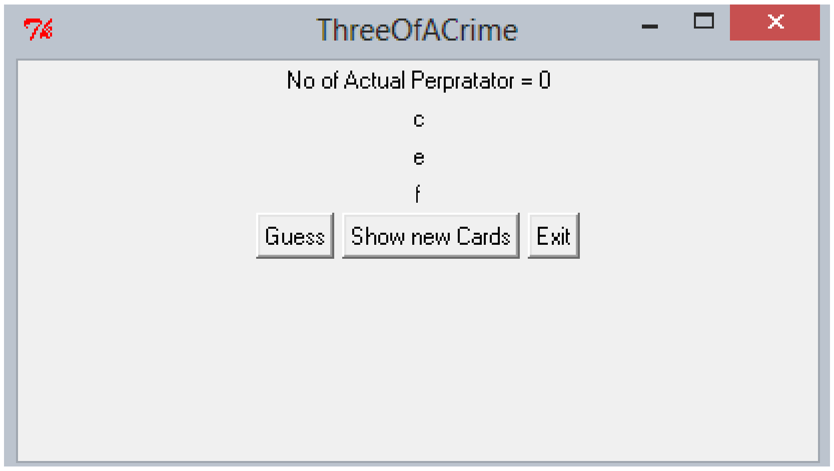
```
	else
	{
		if (($entry_firstGuess -> get() eq $entry_secondGuess-> get()) || ($entry_secondGuess->
get() eq $entry_thirdGuess-> get()) || ($entry_firstGuess -> get() eq $entry_thirdGuess-> get()))
		{
			$label_erorr -> configure(-text=>"All perpetrators must be different\n\n", -
foreground => red);
			$label_erorr -> pack();
		}
		else
		{
			$firstGuess = grep $_ eq $entry_firstGuess -> get(), @criminalNames ; # Search
the Actual perpetrators Array for the user input 1
			$secondGuess = grep $_ eq $entry_secondGuess-> get(), @criminalNames ;#
Search the Actual perpetrators Array for the user input 2
			$thirdGuess = grep $_ eq $entry_thirdGuess-> get(), @criminalNames ; # Search
the Actual perpetrators Array for the user input 3
			if (( $firstGuess == 1 ) && ( $secondGuess == 1 ) && ( $thirdGuess == 1 ) )
			{
				$firstGuess = grep $_ eq $entry_firstGuess -> get(), @ap ; # Search the
Actual perpetrators Array for the user input 1
				$secondGuess = grep $_ eq $entry_secondGuess-> get(), @ap ;# Search
the Actual perpetrators Array for the user input 2
				$thirdGuess = grep $_ eq $entry_thirdGuess-> get(), @ap ; # Search the
Actual perpetrators Array for the user input 3
				if(( $firstGuess == 1 ) && ( $secondGuess == 1 ) && ( $thirdGuess == 1 ) )
# Print the result
					{
					$label_resutl -> configure(-text=>"You Found All Of The Actual
Perpetrators", -foreground => green);
					}
				else
					{
					$label_resutl -> configure(-text=>"Sorry Your answer is Wrong ,
You Loss \n Real Actual Perpetrators : @ap", -foreground => red);
					}
				$frm_guess-> packForget();
				$frm_result-> pack();
			}
			else
			{
				$label_erorr -> configure(-text=>"The perpetrators must be one of
these('a', 'b', 'c', 'd','e', 'f', 'g')\n\n", -foreground => red);
				$label_erorr -> pack();
			}
		}
	}
}
```

Output :

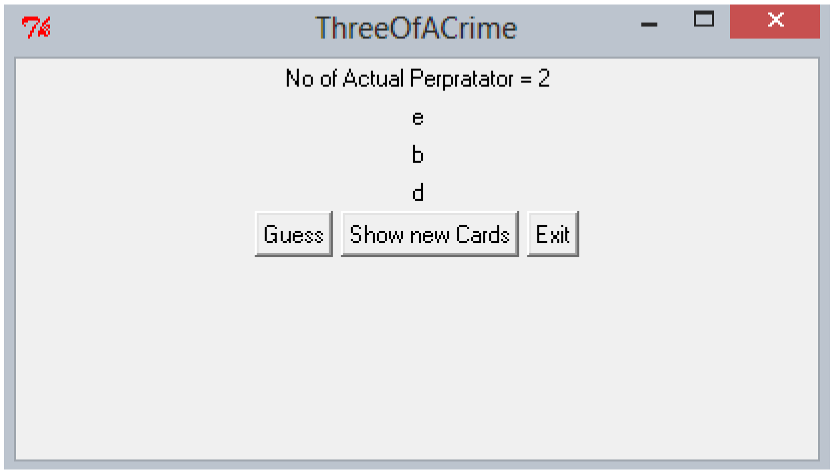


```
No of Actual Perpratator = 1
c
g
d
```

If you press "Show new Cards" Button then it will show you the another card . Following will be the output :

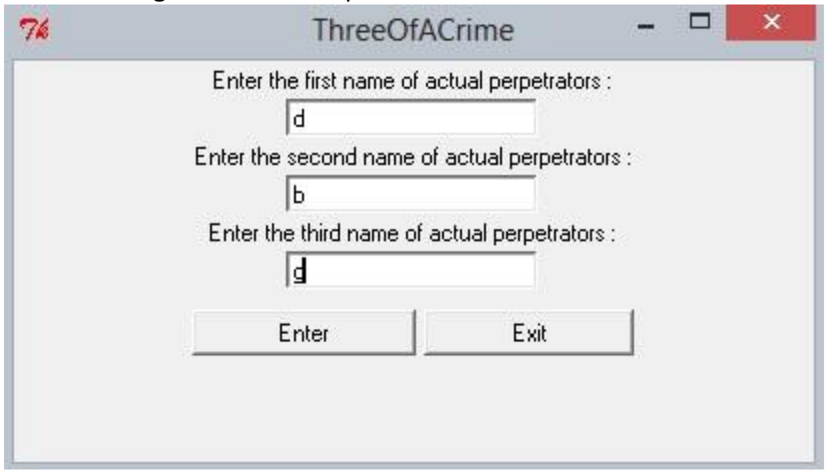


```
No of Actual Perpratator = 0
c
e
f
```



```
No of Actual Perpratator = 2
e
b
d
```

If you press "Guess" Button then it will give you three textbox to enter the Name of actual perpetrator and Following will be the output :


ThreeOfACrime

Enter the first name of actual perpetrators :

d

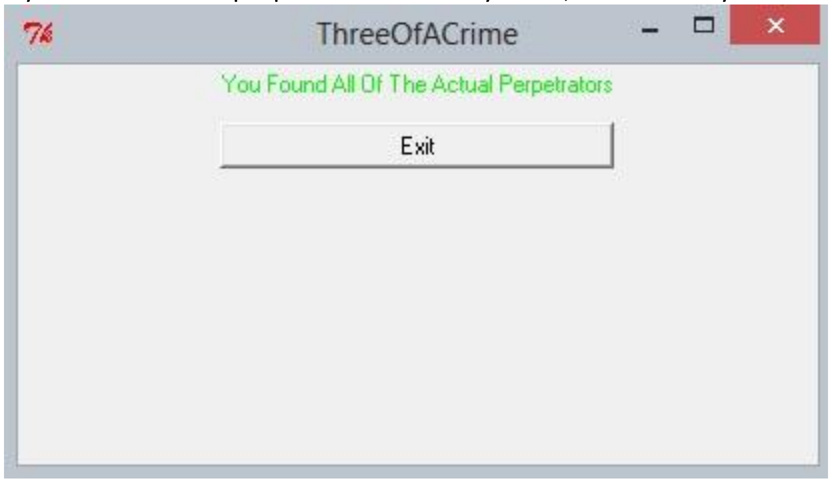
Enter the second name of actual perpetrators :

b

Enter the third name of actual perpetrators :

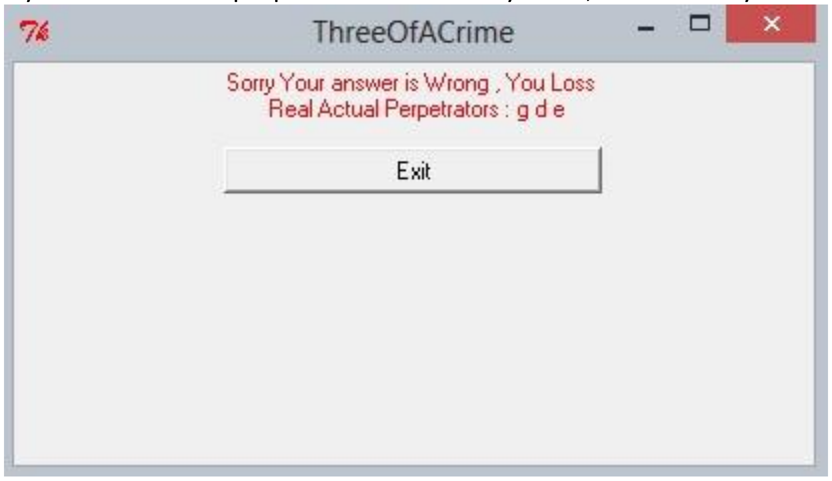
g

If you Guess all the perpetrators correctly then , it will show you the following output :


ThreeOfACrime

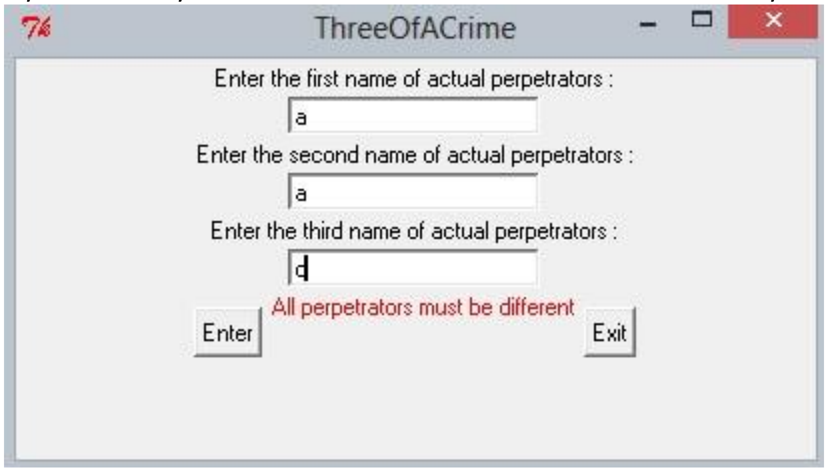
You Found All Of The Actual Perpetrators

If you Guess all the perpetrators incorrectly then , it will show you the following output:


ThreeOfACrime

Sorry Your answer is Wrong , You Loss
Real Actual Perpetrators : g d e

If you enter any two same name in the textbox then it will show you following output :



ThreeOfACrime

Enter the first name of actual perpetrators :

a

Enter the second name of actual perpetrators :

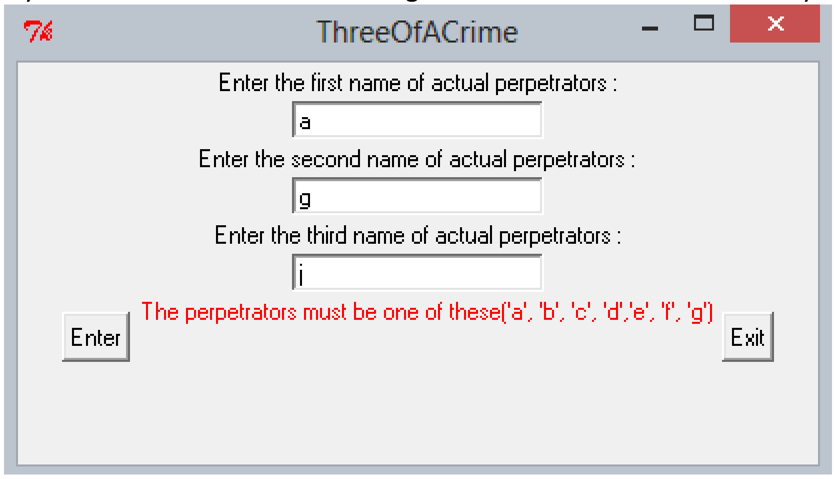
a

Enter the third name of actual perpetrators :

d

All perpetrators must be different

If you enter name other than a to g in the textbox then it will show you following output :



ThreeOfACrime

Enter the first name of actual perpetrators :

a

Enter the second name of actual perpetrators :

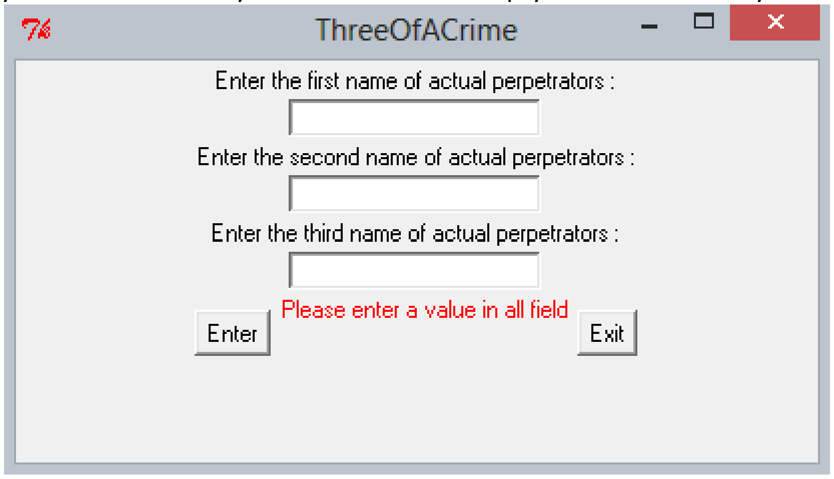
g

Enter the third name of actual perpetrators :

j

The perpetrators must be one of these('a', 'b', 'c', 'd','e', 'f', 'g')

you textboxes of any textbox is remain empty then it will show you following output :



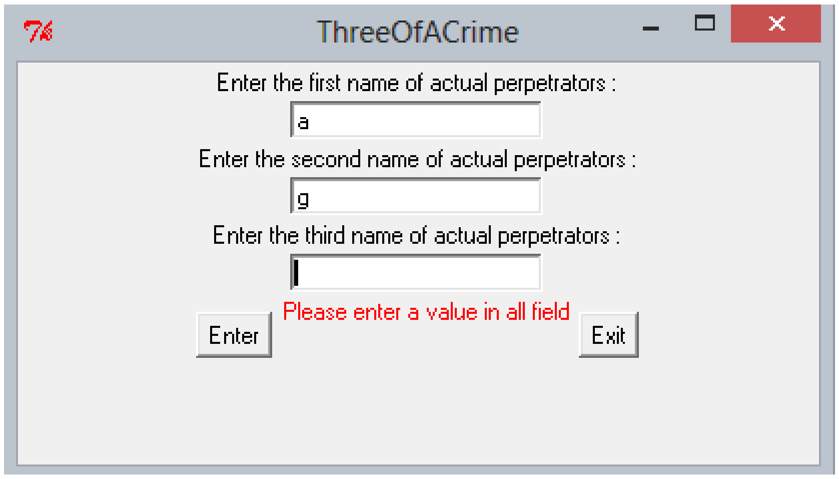
ThreeOfACrime

Enter the first name of actual perpetrators :

Enter the second name of actual perpetrators :

Enter the third name of actual perpetrators :

Please enter a value in all field



ThreeOfACrime

Enter the first name of actual perpetrators :

a

Enter the second name of actual perpetrators :

g

Enter the third name of actual perpetrators :

Please enter a value in all field

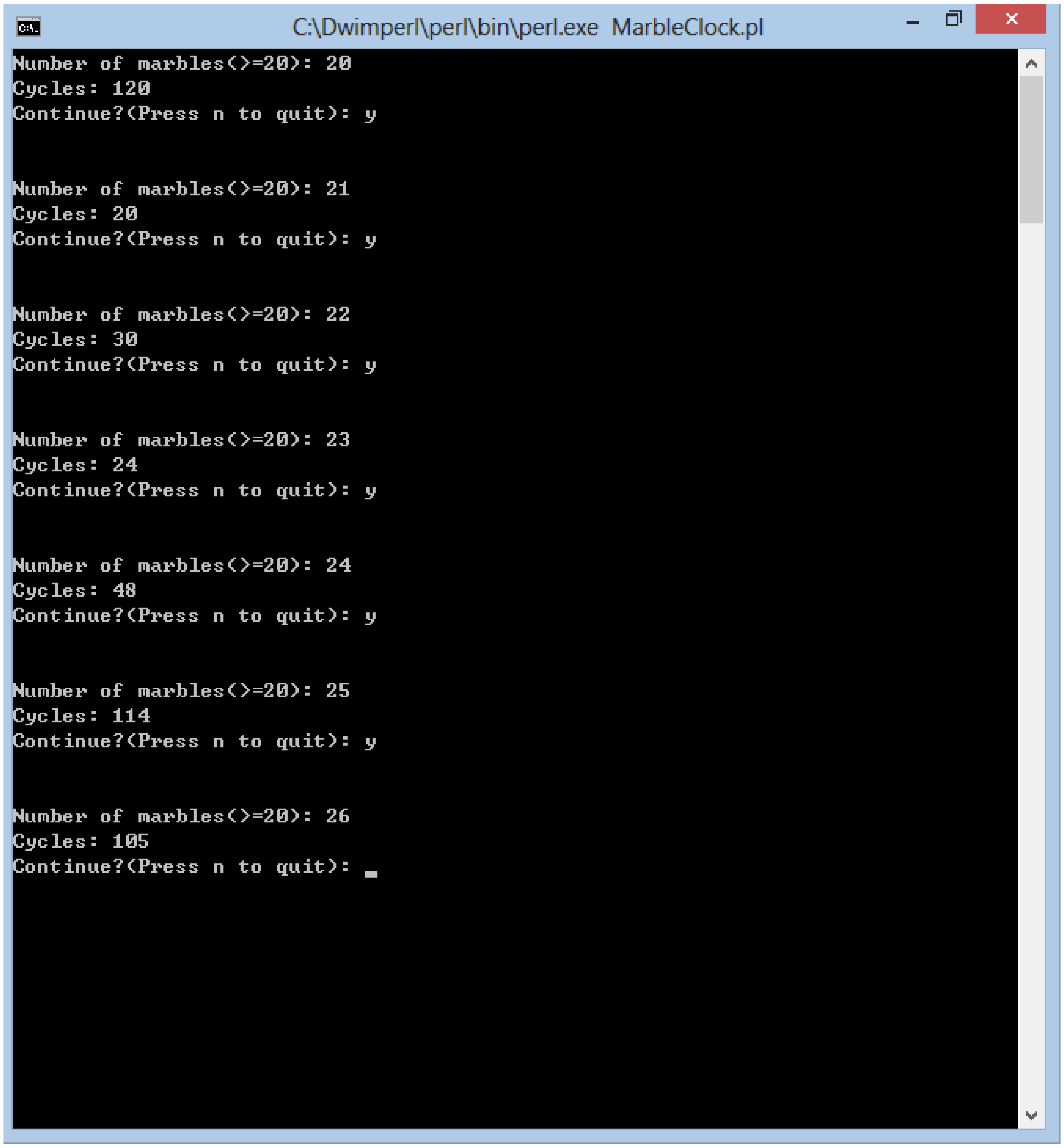
The marble clock

```
#!/usr/bin/perl
# This program demonstrates a working marble clock in Perl
# It produces an iterative cycle for every minute of the Marble CLock
# The user must press enter to print the next iteration

use warnings;

sub printTrays
{
        print "oneMin: ";
        for($i=0;$i<@oneMin;$i++)
        {
                print "$oneMin[$i] ";
        }
        print "\n";

        print "fiveMins: ";
        for($i=0;$i<@fiveMins;$i++)
        {
                print "$fiveMins[$i] ";
        }
        print "\n";

        print "fifteenMins: ";
        for($i=0;$i<@fifteenMins;$i++)
        {
                print "$fifteenMins[$i] ";
        }
        print "\n";

        print "hours: ";
        for($i=0;$i<@hours;$i++)
        {
                print "$hours[$i] ";
        }
        print "\n";

        print "reservoir: ";
        for($i=0;$i<@reservoir;$i++)
        {
                print "$reservoir[$i] ";
        }
        print "\n\n";
}
```

```
#----------------------------------------------------------------------------------------------------------

# assign the default value to $continue
$continue = "y";
# conditional check to let the user determine how many times the marble clock is run

$marbles = 0;
while( $marbles<20 )
 {
          print "Number of marbles(>=20): "; # get input from user.
          $marbles = <>;
 }
 # initialize arrays
@reservoir = ();


for( 1..$marbles ) # enqueues the marbles
 {
          push( @reservoir, $_ )
 }

while( $continue ne "n" )
{
          # initialize number of marbles - runs initial while loop check
          if( @oneMin != 4 ) # length of the @oneMin tray is 4 ( 1, 2, 3, 4 )
          {
                    push( @oneMin, shift( @reservoir ) ); # add to @oneMin tray
                    &printTrays;
          }
          else
          {
                    for($j=0;($j<4)&&($continue ne "n");$j++) # unload 1st tray
                    {
                              push( @reservoir, pop( @oneMin ) );
                              &printTrays;
                              chomp($continue = <>);
                    }

                    if( @fiveMins != 2 ) # length of @fiveMins tray is 2 ( 5 and 10 )
                    {
                              push( @fiveMins , shift( @reservoir ) );
                              &printTrays;
                    }
                    else
                    {
                              for( $j=0;$j<2&&($continue ne "n");$j++ ) # unload second tray
                              {
                                        push( @reservoir, pop( @fiveMins ) );
```

```
                &printTrays;
                chomp($continue = <>);
            }

            if( @fifteenMins != 3 ) # length of @fifteenMins tray is 3 ( 15, 30, 45 )
            {
                push( @fifteenMins, shift( @reservoir ) );
                &printTrays;
            }
            else
            {
                for($j=0;$j<3&&($continue ne "n");$j++) # unload third tray
                {
                    push( @reservoir, pop( @fifteenMins ) );
                    &printTrays;
                    chomp($continue = <>);
                }

                if( @hours != 11 ) # length of @hours tray is 11 ( 1, 2, 3, 4, 5, 6, 7, 8, 9,
10, 11 )
                {
                    push( @hours, shift( @reservoir ) );
                    &printTrays;
                }
                else
                {
                    for($j=0;$j<11&&($continue ne "n");$j++) # unload fourth and
final tray
                    {
                        push( @reservoir, pop( @hours ) );
                        &printTrays;
                        chomp($continue = <>);
                    }
                }
            }
        }
    }

    #print "Continue?(Press n to quit): "; # ask the user to continue or quit
    chomp($continue = <>);
    print "\n";
}
```

Output:


C:\Dwimperl\perl\bin\perl.exe MarbleClock.pl

```
Number of marbles(>=20): 20
Cycles: 120
Continue?(Press n to quit): y


Number of marbles(>=20): 21
Cycles: 20
Continue?(Press n to quit): y


Number of marbles(>=20): 22
Cycles: 30
Continue?(Press n to quit): y


Number of marbles(>=20): 23
Cycles: 24
Continue?(Press n to quit): y


Number of marbles(>=20): 24
Cycles: 48
Continue?(Press n to quit): y


Number of marbles(>=20): 25
Cycles: 114
Continue?(Press n to quit): y


Number of marbles(>=20): 26
Cycles: 105
Continue?(Press n to quit): _
```