Software Construction

echo.0.pl

Perl implementation of /bin/echo always writes a trailing space

```
foreach $arg (@ARGV) {
    print $arg, " ";
}
print "\n";
```

echo.1.pl

Perl implementation of /bin/echo

```
print "@ARGV\n";
```

echo.2.pl

Perl implementation of /bin/echo

```
print join(" ", @ARGV), "\n";
```

sum_arguments.pl

sum integers supplied as command line arguments no check that aguments are numeric

```
$sum = 0;
foreach $arg (@ARGV) {
    $sum += $arg;
}
print "Sum of the numbers is $sum\n";
```

array_growth_demo.pl

```
while (1) {
    print "Enter array index: ";
    $n = <STDIN>;
    if (!$n) {
        last;
    }
    chomp $n;
    $a[$n] = 42;
    print "Array element $n now contains $a[$n]\n";
    printf "Array size is now %d\n", $#a+1;
}
```

line_count.0.pl

Count the number of lines on standard input.

```
$line_count = 0;
while (1) {
    $line = <STDIN>;
    last if !$line;
    $line_count++;
}
print "$line_count lines\n";
```

line_count.1.pl

Count the number of lines on standard input - slightly more concise

```
$line_count = 0;
while (<STDIN>) {
    $line_count++;
}
print "$line_count lines\n";
```

line_count.2.pl

Count the number of lines on standard input - using backwards while to be really concise

```
$line_count = 0;
$line_count++ while <STDIN>;
print "$line_count lines\n";
```

line_count.3.pl

Count the number of lines on standard input. read the input into an array and use the array size.

```
@lines = <STDIN>;
print $#lines+1, " lines\n";
```

line_count.4.pl

Count the number of lines on standard input.
Assignment to () forces a list context and hence reading all lines of input.
The special variable $. contains the current line number

```
() = <STDIN>;
print "$. lines\n";
```

reverse_lines.0.pl

Print lines read from stdin in reverse order.
In a C-style

```
while ($line = <STDIN>) {
    $line[$line_number++] = $line;
}


for ($line_number = $#line; $line_number >= 0 ; $line_number--) {
    print $line[$line_number];
}
```

reverse_lines.1.pl

Print lines read from stdin in reverse order.
Using <> in a list context

```
@line = <STDIN>;
for ($line_number = $#line; $line_number >= 0 ; $line_number--) {
    print $line[$line_number];
}
```

reverse_lines.2.pl

Print lines read from stdin in reverse order.
Using <> in a list context & reverse

```
@lines = <STDIN>;
print reverse @lines;
```

reverse_lines.3.pl

Print lines read from stdin in reverse order.
Using <> in a list context & reverse

```
print reverse <STDIN>;
```

reverse_lines.4.pl

Print lines read from stdin in reverse order.
Using push & pop

```
while ($line = <STDIN>) {
    push @lines, $line;
}
while (@lines) {
    my $line = pop @lines;
    print $line;
}
```

reverse_lines.5.pl

Print lines read from stdin in reverse order.
More succintly with pop

```
@lines = <STDIN>;
while (@lines) {
    print pop @lines;
}
```

reverse_lines.6.pl

Print lines read from stdin in reverse order.
Using unshift

```
while ($line = <STDIN>) {
    unshift @lines, $line;
}
print @lines;
```

cp.0.pl

Simple cp implementation using line by line I/O

```
die "Usage: $0 <infile> <outfile>\n" if @ARGV != 2;

$infile = shift @ARGV;
$outfile = shift @ARGV;

open my $in, '<', $infile or die "Cannot open $infile: $!";
open my $out, '>', $outfile or die "Cannot open $outfile: $!";

while ($line = <$in>) {
    print $out $line;
}

close $in;
close $out;
exit 0;
```

cp.1.pl

Simple cp implementation using line by line I/O relying on the default variable $_

```
die "Usage: $0 <infile> <outfile>\n" if @ARGV != 2;

$infile = shift @ARGV;
$outfile = shift @ARGV;

open my $in, '<', $infile or die "Cannot open $infile: $!";
open my $out, '>', $outfile or die "Cannot open $outfile: $!";

# loop could also be written in one line:
# print OUT while <IN>;

while (<$in>) {
    print $out;
}

close $in;
close $out;
exit 0;
```

cp.2.pl

Simple cp implementation reading entire file into array note that <> returns an array of lines in a list context (in a scalar context it returns a single line)

```
die "Usage: $0 <infile> <outfile>\n" if @ARGV != 2;

$infile = shift @ARGV;
$outfile = shift @ARGV;

open my $in, '<', $infile or die "Cannot open $infile: $!";
@lines = <$in>;
close $in;

open my $out, '>', $outfile or die "Cannot open $outfile: $!";
print $out @lines;
close $out;

exit 0;
```

cp.3.pl

Simple cp implementation via system!
Will break if filenames contain single quotes

```
die "Usage: $0 <infile> <outfile>\n" if @ARGV != 2;

$infile = shift @ARGV;
$outfile = shift @ARGV;

exit system "/bin/cp '$infile' '$outfile'";
```

cp.4.pl

Simple cp implementation reading entire file into array $/ contains the line separator for Perl if it is undefined we can slurp an entire file into a scalar variable with a single read

```
die "Usage: cp <infile> <outfile>\n" if @ARGV != 2;
$infile = shift @ARGV;
$outfile = shift @ARGV;

undef $/;
open my $in, '<', $infile or die "Cannot open $infile: $!";
$contents = <$in>;
close $in;

open my $out, '>', $outfile or die "Cannot open $outfile: $!";
print $out $contents;
close $out;

exit 0;
```

snap_memory.0.pl

Reads lines of input until end-of-input
Print snap! if a line has been seen previously

```
while (1) {
    print "Enter line: ";
    $line = <STDIN>;
    if (!defined $line) {
        last;
    }
    if ($seen{$line}) {
        print "Snap!\n";
    }
    $seen{$line}++;
}
```

snap_memory.1.pl

More concise version of snap_memory.0.pl

```
while (1) {
    print "Enter line: ";
    $line = <STDIN>;
    last if !defined $line;
    print "Snap!\n" if $seen{$line};
    $seen{$line} = 1;
}
```

expel_student.pl

run as ./expel_student mark_deductions.txt find the student with the largest mark deductions expell them

```
while ($line = <>) {
    chomp $line;
    $line =~ s/^"//;
    $line =~ s/"$//;
    my ($name,$offence,$date,$penalty);
    ($name,$offence,$date,$penalty) = split /"\s*,\s*"/, $line;
    $penalty =~ s/[^0-9]//g;
    $deduction{$name} += $penalty;
}

$worst = 0;
foreach $student (keys %deduction) {
    $penalty = $deduction{$student};
    if ($penalty > $worst) {
        $worst_student = $student;
        $worst = $penalty;
    }
}
print "Expel $worst_student who had $worst marks deducted\n";
```

nth_word.pl

Print the nth word on every line of input files/stdin output is piped through fmt to make reading easy

```
die "Usage: $0 <n> <files>\n" if !@ARGV;
$nth_word = shift @ARGV;
open my $f, '|-', "fmt -w 40" or die "Can not run fmt: $!\n";
while ($line = <>) {
    chomp $line;
    @words = split(/ /, $line);
    print $f "$words[$nth_word]\n" if $words[$nth_word];
}
close $f;
```

2d_array.pl

Perl provides only 1 dimensional arrays but arrays elements can contain references to other arrays

```
foreach $i (0..3) {
   foreach $j (0..3) {
        $a[$i][$j] = $i * $j;
    }
}

# We can index @a as if it is a 2d-array
# The following loop prints
#  0  0  0  0
#  0  1  2  3
#  0  2  4  6
#  0  3  6  9

foreach $i (0..3) {
    foreach $j (0..3) {
        printf "%2d ", $a[$i][$j];
    }
    print "\n";
}

# @a contains references to 4 arrays
# the following loop will print something like
# ARRAY(0x55ab77d5e120)
# ARRAY(0x55ab77d5e2a0)
# ARRAY(0x55ab77d687c8)
# ARRAY(0x55ab77d68858)

foreach $i (0..3) {
    print "$a[$i]\n";
}

# We can access the whole array referenced by $a[2] as @{$a[2]}
# the following statement prints
# 0 2 4 6

print "@{$a[2]}\n";
```

using_2d_array.pl

```
@a = ();

# assign reference to array to $a[42]
$a[42] = [1,2,3];

print "$a[42]\n";      # print ARRAY(0x5576c45e8160)
print "@{$a[42]}\n";   # prints 1 2 3

push @{$a[42]}, (4,5,6);
push @{$a[42]}, (7,8,9);

print "$a[42]\n";      # print ARRAY(0x5576c45e8160)
print "@{$a[42]}\n";   # prints 1 2 3 4 5 6 7 8 9/tmp/a.pl
```

COMP(2041|9044) 20T2: Software Construction is brought to you by
the School of Computer Science and Engineering
at the University of New South Wales, Sydney.
For all enquiries, please email the class account at cs2041@cse.unsw.edu.au
CRICOS Provider 00098G