Engineering Computer Network
Beginning Perl
Short Course
http://www.ecn.purdue.edu/~mark/perl
September 25, 2004

Mark Senn
mark@purdue.edu / perl@ecn.purdue.edu

Engineering Computer Network
Purdue University
If you know how to use a text editor and have experience in at least one programming language this short course is designed to try and teach you enough to write simple programs in Perl.
If you know how to use a text editor and have experience in at least one programming language this short course is designed to try and teach you enough to write simple programs in Perl.

Perl is a popular programming language used to write

- bioinformatics software
- system software
- text processing tools
- World Wide Web software
- etc.
If you know how to use a text editor and have experience in at least one programming language this short course is designed to try and teach you enough to write simple programs in Perl.

Perl is a popular programming language used to write

  bioinformatics software
  system software
  text processing tools
  World Wide Web software
  etc.

Larry Wall designed and wrote Perl. The first version was released in 1987. He and others continue to improve it. The decision to do a complete redesign and rewrite of Perl for Perl 6 was made in 2000—that work continues.
Perl is a very rich language. This “Beginning Perl” short course doesn’t cover

- complicated data structures
- modules/packages
- object-oriented programming
- subroutines
- etc.
Perl is a very rich language. This “Beginning Perl” short course doesn’t cover

- complicated data structures
- modules/packages
- object-oriented programming
- subroutines
- etc.

See “Getting More Information” slides for more info on these topics.
Perl is a very rich language. This “Beginning Perl” short course doesn’t cover:

- complicated data structures
- modules/packages
- object-oriented programming
- subroutines
- etc.

See “Getting More Information” slides for more info on these topics. With Perl you don’t need to know anything about the stuff you won’t be using.
Choosing a Programming Language

For projects that involve all programs I used awk, csh, or sh for Perl
## Choosing a Programming Language

<table>
<thead>
<tr>
<th>For projects that involve</th>
<th>I usually use</th>
</tr>
</thead>
<tbody>
<tr>
<td>all programs I used awk, csh, or sh for</td>
<td>Perl</td>
</tr>
<tr>
<td>almost all stuff I used to use C for</td>
<td>Perl</td>
</tr>
</tbody>
</table>
Choosing a Programming Language

For projects that involve
all programs I used awk, csh, or sh for
almost all stuff I used to use C for
character string manipulation
I usually use
Perl
Perl
Perl
## Choosing a Programming Language

<table>
<thead>
<tr>
<th>For projects that involve</th>
<th>I usually use</th>
</tr>
</thead>
<tbody>
<tr>
<td>all programs I used awk, csh, or sh for</td>
<td>Perl</td>
</tr>
<tr>
<td>almost all stuff I used to use C for</td>
<td>Perl</td>
</tr>
<tr>
<td>character string manipulation</td>
<td>Perl</td>
</tr>
<tr>
<td>programs for one-time use</td>
<td>Perl</td>
</tr>
</tbody>
</table>
## Choosing a Programming Language

For projects that involve

<table>
<thead>
<tr>
<th>All programs I used awk, csh, or sh for</th>
<th>I usually use</th>
<th>Perl</th>
</tr>
</thead>
<tbody>
<tr>
<td>Almost all stuff I used to use C for</td>
<td>Perl</td>
<td></td>
</tr>
<tr>
<td>Character string manipulation</td>
<td>Perl</td>
<td></td>
</tr>
<tr>
<td>Programs for one-time use</td>
<td>Perl</td>
<td></td>
</tr>
<tr>
<td>“System tools”</td>
<td>Perl</td>
<td></td>
</tr>
<tr>
<td>Choosing a Programming Language</td>
<td></td>
<td></td>
</tr>
<tr>
<td>----------------------------------</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>For projects that involve</strong></td>
<td><strong>I usually use</strong></td>
<td></td>
</tr>
<tr>
<td>all programs I used awk, csh, or sh for</td>
<td>Perl</td>
<td></td>
</tr>
<tr>
<td>almost all stuff I used to use C for</td>
<td>Perl</td>
<td></td>
</tr>
<tr>
<td>character string manipulation</td>
<td>Perl</td>
<td></td>
</tr>
<tr>
<td>programs for one-time use</td>
<td>Perl</td>
<td></td>
</tr>
<tr>
<td>“system tools”</td>
<td>Perl</td>
<td></td>
</tr>
<tr>
<td>symbolic Mathematics</td>
<td>Mathematica</td>
<td></td>
</tr>
</tbody>
</table>
Choosing a Programming Language

For projects that involve
I usually use
all programs I used awk, csh, or sh for Perl
almost all stuff I used to use C for Perl
character string manipulation Perl
programs for one-time use Perl
“system tools” Perl
symbolic Mathematics Mathematica
calling “canned” FORTRAN subroutines FORTRAN
Choosing a Programming Language

For projects that involve

- all programs I used awk, csh, or sh for
- almost all stuff I used to use C for
- character string manipulation
- programs for one-time use
- “system tools”
- symbolic Mathematics
- calling “canned” FORTRAN subroutines
- lots of numeric calculations

I usually use

- Perl
- Perl
- Perl
- Perl
- Perl
- Mathematica
- FORTRAN
- Mathematica, C or FORTRAN
# Choosing a Programming Language

For projects that involve all programs I used awk, csh, or sh for almost all stuff I used to use C for character string manipulation programs for one-time use “system tools” symbolic Mathematics calling “canned” FORTRAN subroutines lots of numeric calculations low-level software to control hardware

<table>
<thead>
<tr>
<th>Activity</th>
<th>Language</th>
</tr>
</thead>
<tbody>
<tr>
<td>all programs</td>
<td>Perl</td>
</tr>
<tr>
<td>almost all stuff</td>
<td>Perl</td>
</tr>
<tr>
<td>character string manipulation</td>
<td>Perl</td>
</tr>
<tr>
<td>programs for one-time use</td>
<td>Perl</td>
</tr>
<tr>
<td>“system tools”</td>
<td>Perl</td>
</tr>
<tr>
<td>symbolic Mathematics</td>
<td>Mathematica</td>
</tr>
<tr>
<td>calling “canned” FORTRAN subroutines</td>
<td>FORTRAN</td>
</tr>
<tr>
<td>lots of numeric calculations</td>
<td>Mathematica, C or FORTRAN</td>
</tr>
<tr>
<td>low-level software to control hardware</td>
<td>C</td>
</tr>
</tbody>
</table>
Choosing a Programming Language

For projects that involve
all programs I used awk, csh, or sh for
almost all stuff I used to use C for
character string manipulation
programs for one-time use
“system tools”
symbolic Mathematics
calling “canned” FORTRAN subroutines

I usually use
Perl
Perl
Perl
Perl
Perl
Mathematica
FORTRAN
Mathematica, C or FORTRAN
C
whatever we agree upon
This data below is from Diego Zamboni, former Ph.D. student at Computer Sciences. Three different people each wrote a program for generating decision trees from a large dataset in a different language.

<table>
<thead>
<tr>
<th>Language</th>
<th>Development Time</th>
<th>Run Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>≈ 1 week</td>
<td>≈ 2 minutes</td>
</tr>
</tbody>
</table>
This data below is from Diego Zamboni, former Ph.D. student at Computer Sciences. Three different people each wrote a program for generating decision trees from a large dataset in a different language.

<table>
<thead>
<tr>
<th>Language</th>
<th>Development Time</th>
<th>Run Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>$\approx 1$ week</td>
<td>$\approx 2$ minutes</td>
</tr>
<tr>
<td>Java</td>
<td>3–4 days</td>
<td>10 minutes</td>
</tr>
</tbody>
</table>
This data below is from Diego Zamboni, former Ph.D. student at Computer Sciences. Three different people each wrote a program for generating decision trees from a large dataset in a different language.

<table>
<thead>
<tr>
<th>Language</th>
<th>Development Time</th>
<th>Run Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>≈ 1 week</td>
<td>≈ 2 minutes</td>
</tr>
<tr>
<td>Java</td>
<td>3–4 days</td>
<td>≈ 10 minutes</td>
</tr>
<tr>
<td>Perl</td>
<td>1 day</td>
<td>≈ 4 hours</td>
</tr>
</tbody>
</table>
Type `perldoc perldoc` to see information on `perldoc`. `Perldoc` is used to look at Perl documentation based on manual page name, module name, or program name. Type `perldoc perl` to see information regarding Perl itself.
Getting More Information

Type `perldoc perldoc` to see information on `perldoc`. `Perldoc` is used to look at Perl documentation based on manual page name, module name, or program name. Type `perldoc perl` to see information regarding Perl itself.

You may find *Learning Perl*, 2nd Edition, ISBN 1-56592-284-0, by Randal L. Schwartz and Tom Christiansen helpful to learn Perl. Chances are good that if you have a little programming experience you may be able to get by with `perldoc` and/or *Programming Perl*. 
Type `perldoc perldoc` to see information on `perldoc`. `Perldoc` is used to look at Perl documentation based on manual page name, module name, or program name. Type `perldoc perl` to see information regarding Perl itself.

You may find *Learning Perl*, 2nd Edition, ISBN 1-56592-284-0, by Randal L. Schwartz and Tom Christiansen helpful to learn Perl. Chances are good that if you have a little programming experience you may be able to get by with `perldoc` and/or *Programming Perl*.

See http://www.perl.com for pointers to all things Perl.
See http://www.perl.com for pointers to all things Perl.
Perl book information is available at
Getting More Information/2

See http://www.perl.com for pointers to all things Perl.

Perl book information is available at

See http://www.cpan.org for CPAN, the Comprehensive Perl Archive Network. It contains approximately 7000 Perl modules, written by 4000 authors, for a total of 2500 MBytes of code ready for you to use in your programs.
Type \texttt{perl -de 0} (that’s a zero) to start Perl in an interactive mode. This is an easy way to get familiar with Perl. Type Control-D to exit. For example (what to type is underlined):

\begin{verbatim}
% perl -de 0
DB<1> $a = 3
DB<2> $b = $a + 4
DB<3> print "b is $b"
    b is 7
DB<4> [Press Control and d at same time here.]
%
\end{verbatim}
Writing a Perl Program

Put

```perl
#!/usr/bin/perl -w
$| = 1;
```

at the top of your Perl programs.
Writing a Perl Program

Put

```perl
#!/usr/bin/perl -w
$| = 1;
```

at the top of your Perl programs.

The # must be the first character in the file.
Writing a Perl Program

Put

```
#!/usr/bin/perl -w
$| = 1;
```

at the top of your Perl programs.

The # must be the first character in the file.

The -w option will warn you about variables being used before being set, etc.
Put

```perl
#!/usr/bin/perl -w
$| = 1;
```

at the top of your Perl programs.

The `#` must be the first character in the file.

The `-w` option will warn you about variables being used before being set, etc.

“$| = 1” unbuffers standard output.
Comments start with # and go to the end of the line.
General Syntax

Comments start with # and go to the end of the line.

Every statement should be followed by a semicolon (;).
Variables

Variable names may consist of lowercase letters (a–z), uppercase letters (A–Z), and underscore (_).
Variables

Variable names may consist of lowercase letters (a–z), uppercase letters (A–Z), and underscore (_).

There are three main variable types.

- scalars
- arrays
- hashes
A scalar is a single number or string. Its name is prefixed with $.
A scalar is a single number or string. Its name is prefixed with $.

Here are some examples of setting scalars to values:

```perl
$n = 1000;
$pi = 3.1415926535;
$name = "Mark Senn";
$_ = $line;
```
A scalar is a single number or string. Its name is prefixed with $.

Here are some examples of setting scalars to values:

```
$n = 1000;
$pi = 3.1415926535;
$name = "Mark Senn";
$_ = $line;
```

Strings get converted to numbers automatically when doing arithmetic operations.
A scalar is a single number or string. Its name is prefixed with $.

Here are some examples of setting scalars to values:

```perl
$n = 1000;
$pi = 3.1415926535;
$name = "Mark Senn";
$_ = $line;
```

Strings get converted to numbers automatically when doing arithmetic operations.

The default variable is $_. If you don’t explicitly say what variable to operate on, $_ is often used. To specify another variable, use =~, e.g.,

```perl
$var =~ /abc/;
```
An array is a list of values. Each element of an array is a scalar. An array’s name is prefixed with @. Here are some examples of setting array values:

```perl
@prime = (2, 3, 5);
@mixed = (1, 2.3, "word", "more than one word");
@month = ("Jan", "Feb", "Mar");
    # Quote words operator simplifies things.
@month = qw(Jan Feb Mar);
    # Quote words works with stuff on different lines.
@month = qw(
    Jan
    Feb
    Mar
);
    # Copy the values of an entire array.
@b = @a;
```
Array indices start at 0.

To refer to a specific element of an array use, for example,

```
$month[1]
```

In this example that would be Feb.

$\#array$ is the index of the last element of $var$. In this example $\#month$ would be 2.

@array used in a scalar context is how many elements are in array.

@array is always one more than $\#array$. 
A hash is like a regular array but instead of being indexed by integers it is indexed by arbitrary strings. Its name is prefixed with %. About the only time you’ll use the name of a hash with the % is when using the keys function. Usually, you will be manipulating scalar elements of the hash. Here are some examples of setting hash elements to values:

```perl
$office{"Mark"}   = "MSEE 130A";
$office{"George"} = "White House";

# Different syntax to get above results.
%office = (  
    "Mark", "MSEE 130A",  
    "George", "White House"
);
```

Type

```
perldoc perldata
```

or see a Perl book for more information.
Reading/Writing Terminal

Read from the terminal by assigning a variable to `<>`.

```
# Get one line of input from terminal.
$_ = <>;
```
Reading/Writing Terminal

Read from the terminal by assigning a variable to `<>

```perl
$_ = <>;
# Get one line of input from terminal.

chomp;
```

# Get rid of end of line characters.

chomp;
Read from the terminal by assigning a variable to `<>`.

```perl
# Get one line of input from terminal.
$_ = <>;

# Get rid of end of line characters.
chomp;

# Print it.
print "\$_ is \"\$_\"\n";
```
Read from the terminal by assigning a variable to `<>

    # Get one line of input from terminal.
$_ = <>;
    # Get rid of end of line characters.
chomp;
    # Print it.
print "\$_ is \"\$_\"\n";
    # Print same output another way.
print qq/\$_ is "\$_\"\n/;
One manipulates a file by specifying a file name and/or a file handle. I usually use \$fn for the file name and \$FH for the file handle. Note that the file handle is is not preceded by $, @, or %.

    # Define filename.
$fn = "input";
One manipulates a file by specifying a file name and/or a file handle. I usually use $fn for the file name and FH for the file handle. Note that the file handle is not preceded by $, @, or %.

```perl
# Define filename.
$fn = "input";
# "<" opens for input, ">": opens for output.
open FH, "<$fn"
    or die qq/can't open "$fn" for input: $!, stopped/;
```
One manipulates a file by specifying a file name and/or a file handle. I usually use $\text{fn}$ for the file name and $\text{FH}$ for the file handle. Note that the file handle is not preceded by $\$, @, or %.

```perl
# Define filename.
$fn = "input";
# "<" opens for input, ">" opens for output.
open FH, "<$fn"
    or die qq/can't open "$fn" for input: $!, stopped/;
# Read input a line at a time.
while (<FH>) {
```

Reading/Writing Files
One manipulates a file by specifying a file name and/or a file handle. I usually use $fn for the file name and FH for the file handle. Note that the file handle is not preceded by $, @, or %.

```
    # Define filename.
    $fn = "input";
    # "<" opens for input, ">" opens for output.
    open FH, "<$fn"
         or die qq/can’t open "$fn" for input: $!, stopped/;
    # Read input a line at a time.
    while (<FH>) {
        # Get rid of end of line characters.
        chomp;
```
One manipulates a file by specifying a file name and/or a file handle. I usually use $fn for the file name and FH for the file handle. Note that the file handle is not preceded by $, @, or %.

```perl
# Define filename.
$fn = "input";
# "<" opens for input, ">" opens for output.
open FH, "<$fn"
    or die qq/can’t open "$fn" for input: $!, stopped/;
# Read input a line at a time.
while (<FH>) {
    # Get rid of end of line characters.
    chomp;
    # code to process lines goes here
```
One manipulates a file by specifying a file name and/or a file handle. I usually use $fn for the file name and FH for the file handle. Note that the file handle is is not preceded by $, @, or %.

```perl
# Define filename.
$fn = "input";
# "<" opens for input, ">" opens for output.
open FH, "<$fn"
  or die qq/can't open "$fn" for input: $!, stopped/;
# Read input a line at a time.
while (<FH>) {
  # Get rid of end of line characters.
  chomp;
  # code to process lines goes here
}
```
One manipulates a file by specifying a file name and/or a file handle. I usually use $fn for the file name and FH for the file handle. Note that the file handle is not preceded by $, @, or %.

```perl
# Define filename.
$fn = "input";
# "<" opens for input, ">" opens for output.
open FH, "<$fn"
    or die qq/can’t open "$fn" for input: $!, stopped/;
# Read input a line at a time.
while (<FH>) {
    # Get rid of end of line characters.
    chomp;
    # code to process lines goes here
}
close FH
    or die qq/can’t close input file "$fn": $!, stopped/;
```
# Arithmetic Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Operation</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>addition</td>
<td>17 + 3 → 20</td>
</tr>
<tr>
<td>-</td>
<td>subtraction</td>
<td>17 − 3 → 14</td>
</tr>
<tr>
<td>*</td>
<td>multiplication</td>
<td>17 * 3 → 51</td>
</tr>
<tr>
<td>/</td>
<td>division</td>
<td>17/3 → 5.66...</td>
</tr>
<tr>
<td>%</td>
<td>modulo division (remainder)</td>
<td>17 % 3 → 2</td>
</tr>
<tr>
<td>&lt;</td>
<td>less than</td>
<td>17 &lt; 3 → 0</td>
</tr>
<tr>
<td>&lt;=</td>
<td>less than or equal</td>
<td>17 &lt;= 3 → 0</td>
</tr>
<tr>
<td>==</td>
<td>equals</td>
<td>17 == 3 → 0</td>
</tr>
<tr>
<td>!=</td>
<td>not equals</td>
<td>17 != 3 → 1</td>
</tr>
<tr>
<td>&gt;=</td>
<td>greater than or equal</td>
<td>17 &gt;= 3 → 1</td>
</tr>
<tr>
<td>&gt;</td>
<td>greater than</td>
<td>17 &gt; 3 → 1</td>
</tr>
<tr>
<td>&lt;=&gt;</td>
<td>compare</td>
<td>17 &lt;=&gt; 3 → 1</td>
</tr>
</tbody>
</table>

$\text{\$a} \text{ <=> \$b}$ returns −1 if $\text{\$a} < \text{\$b}$, 0 if $\text{\$a} == \text{\$b}$, and 1 if $\text{\$a} > \text{\$b}$
### Bit-wise Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Operation</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>∼1</td>
<td>not</td>
<td>∼1 → all bits 1 except last is 0</td>
</tr>
<tr>
<td></td>
<td>or</td>
<td>1</td>
</tr>
<tr>
<td>&amp;</td>
<td>and</td>
<td>1 &amp; 3 → 1</td>
</tr>
<tr>
<td>^</td>
<td>exclusive or</td>
<td>1 ^ 3 → 2</td>
</tr>
</tbody>
</table>
## Logical Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Operation</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;&amp; or and</td>
<td>and</td>
<td>0 and 1 → 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>or or</td>
</tr>
<tr>
<td>!</td>
<td>not</td>
<td>!4 → 0, !0 → 1</td>
</tr>
<tr>
<td>xor</td>
<td>exclusive or</td>
<td>0 and 1 → 1</td>
</tr>
<tr>
<td>Operator</td>
<td>Operation</td>
<td>Example</td>
</tr>
<tr>
<td>----------</td>
<td>------------------</td>
<td>------------------------------</td>
</tr>
<tr>
<td>.</td>
<td>catenation</td>
<td>&quot;ab&quot; . &quot;cd&quot; → &quot;abcd&quot;</td>
</tr>
<tr>
<td>x</td>
<td>replication</td>
<td>&quot;ab&quot; x 3 → &quot;ababab&quot;</td>
</tr>
<tr>
<td>lt</td>
<td>less than</td>
<td>&quot;ab&quot; lt &quot;cd&quot; → 1</td>
</tr>
<tr>
<td>le</td>
<td>less than or equal</td>
<td>&quot;ab&quot; le &quot;cd&quot; → 1</td>
</tr>
<tr>
<td>eq</td>
<td>equals</td>
<td>&quot;ab&quot; eq &quot;cd&quot; → 0</td>
</tr>
<tr>
<td>ne</td>
<td>not equals</td>
<td>&quot;ab&quot; ne &quot;cd&quot; → 1</td>
</tr>
<tr>
<td>ge</td>
<td>greater than or equal</td>
<td>&quot;ab&quot; ge &quot;cd&quot; → 0</td>
</tr>
<tr>
<td>gt</td>
<td>greater than</td>
<td>&quot;ab&quot; gt &quot;cd&quot; → 0</td>
</tr>
<tr>
<td>cmp</td>
<td>compare</td>
<td>&quot;ab&quot; cmp &quot;cd&quot; → -1</td>
</tr>
</tbody>
</table>

$a$ cmp $b$ returns -1 if $a$ lt $b$, 0 if $a$ eq $b$, and 1 if $a$ gt $b
If \textit{expression} is true then do \textit{statement1} and \textit{statement2}. Note that the braces are necessary even if you just have one statement.

\textbf{General Form} \hspace{1cm} \textbf{Example}

if (expression) {
    statement1;
    print "it’s hot!\n";
    statement2;
}

if ($\text{temp} > 100) {
    \text{print } "it seems hot!\n";
}

if ($\text{t} > 100 \text{ or } ($\text{t} > 80 \text{ and } $\text{h} > 80) ) {
    \text{print } "it seems hot!\n";
}

if ($\text{t} > 100 \text{ or } ($\text{t} > 80 \text{ and } $\text{h} > 80) ) {
    \text{print } "it seems hot!\n";
}
If \( expression \) is true then do \( statement1 \) and \( statement2 \) otherwise (\( expression \) was false) and \( statement3 \) and \( statement4 \) will be done. Note that the braces are necessary even if you just have one statement.

**General Form**

\[
\text{if (expression)} \ { \{ \begin{align*}
\text{statement1; } \\
\text{statement2; }
\end{align*} \} \ } \text{ } \text{else} \ { \{ \begin{align*}
\text{statement3; } \\
\text{statement4; }
\end{align*} \} }
\]

**Example**

\[
\text{if ($temp > 100) } \ { \{ \begin{align*}
\text{print "it’s hot\n"; } \\
\text{else } \ { \{ \begin{align*}
\text{print "it’s not so hot\n"; }
\end{align*} \} }
\end{align*} \} }
\]

\[
\text{if (grep /^$day$/, qw(Sat Sun)) } \ { \{ \begin{align*}
\text{print "it’s a weekend\n"; } \\
\text{else } \ { \{ \begin{align*}
\text{print "it’s a weekday\n"; }
\end{align*} \} }
\end{align*} \} }
\]
Three common types of loops are described here: for, foreach, and while. Perl also has continue, do, goto, labels, unless, and until loop-related statements.

Type perldoc perlsyn or see a Perl book for more information.

For each type of loop redo goes back to the top of the loop, next goes to the bottom of the loop, and last breaks out of the loop. See below.
General Form

\begin{verbatim}
for (init; expression; incr) {
    # redo comes to here
    statement1;
    statement2;
    # next comes to here
    # (the incr gets done here)
}
# last comes to here
\end{verbatim}

Example

\begin{verbatim}
# Print numbers 1-20
# except for 13.
for ($i=1; $i<=20; $i++) {
    (13 == $i) and next;
    print "$i
";
}
# Double each element of @a.
for ($i=0; $i<@a; $i++) {
    $a[$i] = 2 * $a[$i];
}
\end{verbatim}
**General Form**

while (expression) {
    # redo comes to here
    statement1;
    statement2;
    # next comes to here
}
# last comes to here

**Examples**

# Print numbers 1-20 except for 13.
$i = 1;
while ($i <= 20) {
    (13 == $i) and next;
    print "$i\n";
    $i++;
}

# Read and print lines.
while (<>) {
    print qq/line read was "$_"\n/;
}

# Add x’s to on end of 
while (length < $t) {
    $$_ = $$_ . "x";
}
Perl has *lots* of functions. Type `perldoc perlfuns` or see a Perl book for more information.

Functions can be called using

```
function_name(argument1, argument2, ...)
```

or

```
function_name argument1, argument2, ...
```

The second form is much more common.
Here are a few of Perl's built-in functions:

chomp string
Delete end of line character(s) from string and return number of characters deleted as value of function. Note that string can’t be a constant (e.g., "string") because there wouldn’t be anywhere to put the shortened string.

close filehandle
Close a filehandle that was previously opened with open.

die message
Print “message at program line line” and abort program. For example, if $fn is set to “f” in program “p”
die qq/Can’t open "$fn": $!, stopped/;
might print
Can’t open "f": permission denied, stopped at p line 4.
grep *expression*, *array*

For each element of *array*, set \$_ to that element, and evaluate *expression*. If value is used in an array context return all elements of *array* for which *expression* was true. If value is used in a scalar context return number of times *expression* was true.

**keys *hash***

Returns an array consisting of all of *hash*’s keys in no particular order. In $name{\$k} = \$v$, \$k* is a key and \$v* is a value. (See “values.”)

**length *string***

Returns number of characters in *string*. 
open \texttt{filehandle, expression}

Open a file. Type \texttt{perldoc perlopentut}
or see a Perl book for more information.

\texttt{pop array}

Delete last element from \texttt{array} and return deleted element as value of function. (See \texttt{push}.)

\texttt{push array, scalar}

Adds \texttt{scalar} to the end of \texttt{array}.

\texttt{sort array}

Return sorted \texttt{array} as value of function.

\texttt{values hash}

Returns an array consisting of all of \texttt{hash}'s values in no particular order. (See \texttt{keys}.)
Regular Expressions

Regular expressions are used to match strings.
Regular Expressions

Regular expressions are used to match strings.

There are *lots* of things you can put in a regular expressions.
Regular expressions are used to match strings.

There are *lots* of things you can put in a regular expressions.

This is a *simple* and *incomplete* introduction.
Regular Expressions

Regular expressions are used to match strings.
There are lots of things you can put in a regular expressions.
This is a simple and incomplete introduction.
Type perldoc perlrere or see a Perl book for more information.
A character class is a set of characters. It is used to match any of the characters in the character class.

A character class starts with [ and ends with ].
A character class is a set of characters. It is used to match any of the characters in the character class.

A character class starts with [ and ends with ]. In between put the characters you want matched.
Character Classes

A character class is a set of characters. It is used to match any of the characters in the character class.

A character class starts with [ and ends with ]. In between put the characters you want matched.

For example, \[abcn-z\] matches a or b or c or any of the characters n through z.
Character Classes

A character class is a set of characters. It is used to match any of the characters in the character class.

A character class starts with [ and ends with ]. In between put the characters you want matched.

For example, \[\text{abcn-z}\] matches a or b or c or any of the characters n through z.

To include the - character in the character class make it the first character. Example: \[\text{-abcn-z}\]
A character class is a set of characters. It is used to match any of the characters in the character class.

A character class starts with [ and ends with ]. In between put the characters you want matched.

For example, [abcn-z] matches a or b or c or any of the characters n through z.

To include the – character in the character class make it the first character. Example: [-abcn-z]

If the first character after the [ is ^ (uparrow) that means match all characters not in the character class.
A character class is a set of characters. It is used to match any of the characters in the character class.

A character class starts with [ and ends with ]. In between put the characters you want matched.

For example, [abcn-z] matches a or b or c or any of the characters n through z.

To include the - character in the character class make it the first character. Example: [-abcn-z]

If the first character after the [ is ^ (uparrow) that means match all characters not in the character class. Example: [^-+*\/]
### Matching a single character

<table>
<thead>
<tr>
<th>Use This</th>
<th>To Match</th>
</tr>
</thead>
<tbody>
<tr>
<td>.</td>
<td>any character except newline (\n)</td>
</tr>
<tr>
<td>\d</td>
<td>a digit ([0-9])</td>
</tr>
<tr>
<td>\D</td>
<td>a non-digit ([^0-9])</td>
</tr>
<tr>
<td>\f</td>
<td>form feed (Control-L, ASCII decimal 12)</td>
</tr>
<tr>
<td>\n</td>
<td>newline (Control-J, ASCII decimal 10)</td>
</tr>
<tr>
<td>\r</td>
<td>carriage return (Control-M, ASCII decimal 13)</td>
</tr>
<tr>
<td>\s</td>
<td>a space character ([ \r\t\n\f])</td>
</tr>
<tr>
<td>\S</td>
<td>a non-space character ([^ \r\t\n\f])</td>
</tr>
<tr>
<td>\t</td>
<td>tab (Control-I, ASCII decimal 9)</td>
</tr>
<tr>
<td>\w</td>
<td>a word character ([a-zA-Z0-9_])</td>
</tr>
<tr>
<td>\W</td>
<td>a non-word character ([^a-zA-Z0-9_])</td>
</tr>
<tr>
<td>\</td>
<td>most anything else</td>
</tr>
<tr>
<td>\</td>
<td>it</td>
</tr>
</tbody>
</table>
Matching a sequence of characters

To match a sequence of characters just put the sequence of characters you want to match in the regular expression.

For example, putting `ab\d` in a regular expression would match `ab` followed by a digit.
To group something enclose it in parentheses. You can then specify a repeat count for what's in parentheses and/or refer to what was matched later by using $1, $2, . . . .
To group something enclose it in parentheses. You can then specify a repeat count for what’s in parentheses and/or refer to what was matched later by using $1, $2, ....

In parentheses you can put alternatives separated by |. The regular expression

```
/([aeiou]|1234)/
```

will match a single vowel or the string 1234. If there was a match, exactly what it matched will be in $1.
Repetition Counts

To match a character or group multiple times put a repetition count after the character or group specification:

<table>
<thead>
<tr>
<th>Repetition Count</th>
<th>Matches</th>
</tr>
</thead>
<tbody>
<tr>
<td>?</td>
<td>0 or 1 times</td>
</tr>
</tbody>
</table>
Repetition Counts

To match a character or group multiple times put a repetition count after the character or group specification:

<table>
<thead>
<tr>
<th>Repetition Count</th>
<th>Matches</th>
</tr>
</thead>
<tbody>
<tr>
<td>?</td>
<td>0 or 1 times</td>
</tr>
<tr>
<td>*</td>
<td>0 or more times</td>
</tr>
</tbody>
</table>
Repetition Counts

To match a character or group multiple times put a repetition count after the character or group specification:

<table>
<thead>
<tr>
<th>Repetition Count</th>
<th>Matches</th>
</tr>
</thead>
<tbody>
<tr>
<td>?</td>
<td>0 or 1 times</td>
</tr>
<tr>
<td>*</td>
<td>0 or more times</td>
</tr>
<tr>
<td>+</td>
<td>1 or more times</td>
</tr>
</tbody>
</table>
Repetition Counts

To match a character or group multiple times put a repetition count after the character or group specification:

<table>
<thead>
<tr>
<th>Repetition Count</th>
<th>Matches</th>
</tr>
</thead>
<tbody>
<tr>
<td>?</td>
<td>0 or 1 times</td>
</tr>
<tr>
<td>*</td>
<td>0 or more times</td>
</tr>
<tr>
<td>+</td>
<td>1 or more times</td>
</tr>
<tr>
<td>{m,n}</td>
<td>at least (m) but no more than (n) times</td>
</tr>
</tbody>
</table>
### Repetition Counts

To match a character or group multiple times put a repetition count after the character or group specification:

<table>
<thead>
<tr>
<th>Repetition Count</th>
<th>Matches</th>
</tr>
</thead>
<tbody>
<tr>
<td>?</td>
<td>0 or 1 times</td>
</tr>
<tr>
<td>*</td>
<td>0 or more times</td>
</tr>
<tr>
<td>+</td>
<td>1 or more times</td>
</tr>
<tr>
<td>{m,n}</td>
<td>at least (m) but no more than (n) times</td>
</tr>
<tr>
<td>{m,}</td>
<td>at least (m) times</td>
</tr>
</tbody>
</table>
To match a character or group multiple times put a repetition count after the character or group specification:

<table>
<thead>
<tr>
<th>Repetition Count</th>
<th>Matches</th>
</tr>
</thead>
<tbody>
<tr>
<td>?</td>
<td>0 or 1 times</td>
</tr>
<tr>
<td>*</td>
<td>0 or more times</td>
</tr>
<tr>
<td>+</td>
<td>1 or more times</td>
</tr>
<tr>
<td>{m,n}</td>
<td>at least (m) but no more than (n) times</td>
</tr>
<tr>
<td>{m,}</td>
<td>at least (m) times</td>
</tr>
<tr>
<td>{,n}</td>
<td>no more than (n) times</td>
</tr>
</tbody>
</table>
Beginning and Ending of Lines

Putting a ^ at the beginning of a regular expression forces the match to start at the beginning of a line.
Putting a `^` at the beginning of a regular expression forces the match to start at the beginning of a line.

Putting a `$` as the last character of a regular expression forces the match to end at the end of a line.
if (/ab/) {
    print "\"ab\" is in \"$_\"\n";
}

Regular Expression Examples

if (/ab/) {
    print "\"ab\" is in \"$_\"\n";
}

# Same as previous example except use qq
# operator to "double quote" so we don’t
# have to use \" inside double quoted string.
if (/ab/) {
    print qq/"ab" is in "$_"./\n/;
}
Regular Expression Examples

if (/ab/) {
    print "\"ab\" is in \"$_\"\n";
}

# Same as previous example except use qq
# operator to "double quote" so we don’t
# have to use " inside double quoted string.
if (/ab/) {
    print qq/"ab" is in "$_"./\n/;
}

    # Same as previous example.
/ab/ and print qq/"ab" is in "$_"\n/;
if ($str =~ /yz$/) {
    print qq/"yz" at end of line in "$str"./;
}

if ($str =~ /yz$/) {
    print qq/
"yz" at end of line in "$str".
/;
}

# Same as previous example.
($str =~ /yz$/)
    and print qq/
"yz" at end of line in "$str".
/;
if ($str =~/yz$/) {
    print qq/"yz" at end of line in "$str".\n/;
}

# Same as previous example.
($str =~/yz$/)
    and print qq/"yz" at end of line in "$str".\n/;

# Set $match to 1 if second character
# on line of $str is "X", 0 otherwise.
$match = ($str =~/\^X/);
The Substitution Operator

The general form of the substitution operator is \texttt{s/old/new/}. By default the substitution is done on \$\_\_. To do it on another scalar do, for example:

\begin{verbatim}
$home =~s/DC/TX/;  # when done being president
\end{verbatim}

Here are some examples of the substitution command:

<table>
<thead>
<tr>
<th>Command</th>
<th>$_ before</th>
<th>$_ after</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{s/abc/def/;}</td>
<td>xyz</td>
<td>xyz</td>
<td>no change</td>
</tr>
<tr>
<td>\texttt{s/abc/def/;}</td>
<td>abc</td>
<td>def</td>
<td>found it</td>
</tr>
<tr>
<td>\texttt{s/abc/def/;}</td>
<td>abcabc</td>
<td>defabc</td>
<td>only first done</td>
</tr>
<tr>
<td>\texttt{s/abc/def/g;}</td>
<td>abcabc</td>
<td>defdef</td>
<td>/g does it globally</td>
</tr>
<tr>
<td>\texttt{s/a*bc/bc/;}</td>
<td>abcabc</td>
<td>bcabc</td>
<td>repeat count after a</td>
</tr>
<tr>
<td>\texttt{s/a*bc/bc/g;}</td>
<td>abcabc</td>
<td>bcbc</td>
<td>/g does it globally</td>
</tr>
<tr>
<td>\texttt{s/^a/A/;}</td>
<td>abcabc</td>
<td>Abcabc</td>
<td>^ matches beginning of line</td>
</tr>
<tr>
<td>\texttt{s/c$//C/;}</td>
<td>abcabc</td>
<td>abcabC</td>
<td>$ matches end of line</td>
</tr>
<tr>
<td>\texttt{s/[ac]/x/;}</td>
<td>abcabc</td>
<td>xbcabc</td>
<td>character class for a or c</td>
</tr>
</tbody>
</table>
Here are some examples of the substitution command:

- `s/[ac]/x/g; abcabc xbxxbx`  
  previous line plus /g for global

- `s/[a-c]/x/g; abcabc xxxxxx`  
  character class for a–c

- `s/a//; abcabc bcabc`  
  substituting with nothing

- `s/.//; abcabc bcabc`  
  . matches anything

- `s/^.(.*).$/\1/; abcabc bcab`  
  delete first and last characters of line

- `s/\d*$//; abc123 abc`  
  delete trailing digits

- `s#//home/##/; /home/abc abc`  
  don’t use have to use /

For more information type `perldoc perlop` or see a Perl book.
Putting a Perl Program on the Command Line

Type

perl -e 'expression'

to evaluate Perl expressions from the command line. Small Perl programs can be run this way.
Putting a Perl Program on the Command Line

Type

perl -e 'expression'

to evaluate Perl expressions from the command line. Small Perl programs can be run this way.

Type

perldoc perlr

or see a Perl book for more information.