9.1 Origins and Uses of Perl
- Began in the late 1980s as a more powerful replacement for the capabilities of awk (text file processing) and sh (UNIX system administration)
- Now includes sockets for communications and modules for OOP, among other things
- Now the most commonly used language for CGI, in part because of its pattern matching capabilities
- Perl programs are usually processed the same way as many Java programs, compilation to an intermediate form, followed by interpretation

9.2 Scalars and Their Operations
- Scalars are variables that can store either numbers, strings, or references (discussed later)
- Numbers are stored in double format; integers are rarely used
- Numeric literals have the same form as in other common languages

9.2 Scalars and Their Operations (continued)
- Perl has two kinds of string literals, those delimited by double quotes and those delimited by single quotes
- Single-quoted literals cannot include escape sequences
- Double-quoted literals can include them
- In both cases, the delimiting quote can be embedded by preceding it with a backslash
- If you want a string literal with single-quote characteristics, but don't want delimit it with single quotes, use qx, where x is a new delimiter
  - For double quotes, use qx
  - If the new delimiter is a parenthesis, a brace, a bracket, or a pointed bracket, the right delimiter must be the other member of the pair
- A null string can be '' or ""

9.2 Scalars and Their Operations (continued)
- Scalar type is specified by preceding the name with a $
- Name must begin with a letter; any number of letters, digits, or underscore characters can follow
- Names are case sensitive
- By convention, names of variables use only lowercase letters

9.3 Assignment Statements and Simple Input and Output
- Assignment statements are as those in C++ & Java
- All Perl statements except those at the end of blocks must be terminated with semicolons
- Comments are specified with #
- Keyboard Input
  - Files are referenced in Perl programs with filehandles
  - STDIN is the predefined filehandle for standard input, the keyboard
  - The line input operator is specified with <
    - If the input is a string value, we often want to trim off the trailing newline, so we use chomp($new = <STDIN>);
  - Files are referenced in Perl programs with filehandles
  - STDIN is the predefined filehandle for standard input, the keyboard
  - The line input operator is specified with <
    - If the input is a string value, we often want to trim off the trailing newline, so we use chomp($new = <STDIN>);
9.3 Assignment Statements and Simple Input and Output (continued)

- Screen Output
  
  ```
  print one or more string literals, separated by commas
  
  print "The answer is $result 
  
  Example program:
  
  print "Please input the circle's radius: ";
  $radius = <STDIN>
  $area = 3.14159265 * $radius * $radius;
  print "The area is: $area 
  
  One way to run a Perl program:
  
  perl prog1.pl
  ```

- Two useful flags:
  - `>` means compile only (for error checking)
  - `w` means produce warnings for suspicious stuff (you should always use this!)

- To get input from a file (read with `<>`):
  ```
  perl prog1.pl prog1.dat
  ```

9.4 Control Statements

- Control Expressions
  
  1. Scalar-valued expressions
     - If it's a string, it's true unless it is either the null string or it's "0"
     - If it's a number, it's true unless it is zero
  
  2. Relational Expressions
     - Operands can be any scalar-valued expressions
       
       | Numeric Operands | String Operands |
       |------------------|-----------------|
       | `==` eq          | `!=` ne         |
       | `>` gt          | `>=` ge         |
       | `<=` le         |                 |
       
     - If a string operator gets a numeric operand, the
       operand is coerced to a string; likewise for numeric operators

- Boolean Expressions
  
  - Operators:
    ```
    &&, ||, ! (higher precedence), as well as and, or, and not (lower precedence)
    ```
  
  - See Table 9.4, p. 345, for the precedence and the associativity of operators
  
  - Assignment statements have values, so they can be
    used as control expressions
      ```
      while ($next = <STDIN>) …
      ```
  
  - The keyboard EOF is specified with:
    ```
    Control+D for UNIX
    Control+z for Windows
    COMMAND+. For Macintosh
    ```

9.4 Control Statements (continued)

- Selection Statements
  ```
  if (control expression) { 
    then-clause
  }
  
  elsif clauses can be included

  unless (control expression) {
    unless-clause
  }
  ```
  
  - Braces are not required
  - Unless clauses can be included
    ```
    while (<STDIN>) {
      print;
      chomp;
      if ($_ eq "gold") {
        print "I'm rich, I'm rich!!! 
      }
    }
    ```

- Loop Statements
  ```
  while (control expression) { 
    loop-body
  }
  ```
  
  - The initial and increment expressions can be
    'comma' expressions
  
  - Switch - Perl does not have one
  
  - Can be built with the `last` operator, which
    transfers control out of the block whose label
    is given as its operand
  
  ```
  SWITCH: {  # SWITCH is the block label
    if ($input eq "bunny") {
      $rabbit++;
      last SWITCH;
    } else if ($input eq "puppy") {
      $dog++;
      last SWITCH;
    } else {
      print "$input is neither a bunny",
      " nor a puppy " 
    }
  }
  ```

- The implicit variable `$` is used as the default
  operand for operators and the default parameter
  in function calls
  ```
  while (<STDIN>) {
    print;
    chomp;
    if ($_, eq "gold") { 
      print "I'm rich, I'm rich!!! 
    }
  }
  ```

9.5 Fundamentals of Arrays

- Perl arrays store only scalar values, which can store strings, numbers, and references
  ```
  while (<STDIN>) { 
    print;
    chomp;
    if ($_, eq "gold") { 
      print "I'm rich, I'm rich!!! 
    }
  }
  ```

- A list is an ordered sequence of scalar values
- A list literal is a parenthesized list of scalar expressions
  ```
  Examples:
  "Apple", $sum / $quantity, 2.732e-21
  ```
9.5 Fundamentals of Arrays (continued)

- An array is a variable that can store a list
- Array names all begin with at signs (@)
- Arrays can be assigned other arrays or list literals
  
  ```perl
glist = (1, 4, 8);
glist2 = @list;
```
- If an array is used where a scalar is expected, the length of the array is used
  
  ```perl@glist = (1, 55, 193);
$len = @list;  # $len now has the value 3```
- A list literal that has only scalar names can be the target of a list assignment
  
  ```perl$(one, $two, $three) = (1, 2, 3);
```
- When an array element is referenced or assigned, the name is a scalar name
  
  ```perl@glist[3] = 17;
$age = $list[1];```
- The length of an array is dynamic; it is always the highest subscript that has been assigned, plus 1 (It is NOT necessarily the number of elements)

9.5 Fundamentals of Arrays (continued)

- The last subscript of an array is its name, preceded by @
- This value can be assigned
- Scalar context versus list context
  - Often forced by an operator or a function
  - Scalar context can be forced with the scalar function
  - The foreach statement - to process arrays and hashes
    ```perl
    foreach $price (@price_list) {
    $price += 0.20;
    }
    ```
- The foreach variable acts as an alias for the elements of the array
- List Operators
  - shift - removes and returns the first element of its list operand
    ```perl$sleft = shift @list;
```
  - unshift - puts its second operand (a scalar or a list) on the left end of its first operand (an array)
    ```perl
   .unshift @list, 47;
    ```
  - pop - a shift on the right end
  - push - an unshift of the right end
  - split - breaks strings into parts using a specific character as the split character
    ```perl
    $stuff = "233:466:688";
    $numbers = split /:/, $stuff;
    ```
  - sort - sorts using string comparisons (numbers are coerced to strings)
    ```perl
    $numbers = sort $numbers;
    ```
  - die - like print, except it also kills the program
    ```perl
    die "Error: division by zero in function fun2"
    ```

# process_names.pl - A simple program to
#         illustrate the use of arrays
#  Input: A file, specified on the command
#         line, of lines of text, where each
#         line is a person's name
# Output: The input names, after all letters
#         are converted to uppercase, in
#         alphabetical order

```
$index = 0;
# Loop to read the names and process them
while($name = <>) {
  # Convert the names' letters to uppercase
  # and put it in the names array
  $names{$index++} = uc($name);
}
# Display the sorted list of names
print "The sorted list of names is:


";
```
```
foreach $name (sort @names) {
  print "$name 
";
}
```

9.5 Fundamentals of Arrays (continued)

# Differ from arrays in two fundamental ways:
1. Arrays use numerics as indices, hashes use strings
2. Array elements are ordered, hash elements are not
   - Hash names begin with percent signs (%)
   - List literals are used to initialize hashes
   - Can be comma-separated values, as in
     ```perl
     @hash = ("Monday", 10451, "Tuesday", 10580);
     ```
   - Or, implication symbols can be used between a key and its value, as in
     ```perl
     %hash = (
     "Monday" => 10451,
     "Tuesday" => 10580;
     )
     ```
   - The left operand of => need not be quoted
   - Subscripts are keys (strings) placed in braces
     ```perl
     @hash = %hash;
     ```
   - Keys and values can be moved from a hash to an array with keys and values
     ```perl
     foreach $name (keys %hash) {
     print "Salary of $name is: $salaries{$name} 
";
     }
     ```
   - Perl has a predefined hash named %ENV, which stores operating system environment variables and their values (see Chapter 10)
9.7 References

- A reference is a scalar variable that references another variable or a literal
- A reference to an existing variable is obtained with the backslash operator
  \$ref_sum = \$sum;
- A reference to a list literal is created by placing the literal in brackets
  \$ref_list = [2, 4, 6, 8];
- A reference to a hash literal is created by placing the literal in braces
  \$ref_hash = \{ Mom => 47, Dad => 48 \};
- All dereferencing in Perl is explicit
  - For scalars, add a $ to the beginning
  - For arrays and hashes,
    1. Add a $ to the beginning of the name, or
    2. Put the -> operator between the name and its subscript
      \$ref_hash -> ("Mom") = 48;

9.8 Functions

- A function definition is the function header and a block of code that defines its actions
- A function header is the reserved word sub and the function's name
- A function declaration is a message to the compiler that the given name is a function that will be defined somewhere in the program
  - Syntactically, a function declaration is just the function's header
  - Function definitions can appear anywhere in a program
  - Function calls can be embedded in expressions (if they return something useful) or they can be standalone statements (if they don't)
  - A function that has been previously declared can be treated as a list operator
  - A function can specify a return value in two ways:
    1. As the operand of a return statement (a function can have zero or more returns)
    2. As the value of the last evaluated expression in the function

9.8 Functions (continued)

sub fun1 {
  my($first) = \$_[0];
  ++$first * ++\$_[1];
}

- Pass-by-reference parameters can be implemented by passing references

sub sub1 (
  my($ref_len, $ref_list) = \$_;
  my $count;
  for ($count = 0; $count < $$ref_len;
    $$ref_list[$count++]);
}

- An example call to sub1:
  sub1($len, \@mylist);

sub median {
  my $len = \$_[0];
  my @list = \$_;
  # Discard the first element of the array
  shift(@list);
  # Sort the parameter array
  @list = sort @list;
  # Compute the median
  if ($len % 2 == 1) {  # length is odd
    return \$list[$len / 2];
  } else {  # length is even
    return (\$list[$len / 2] + \$list[$len / 2 - 1]) / 2;
  }
}

9.8 Functions (continued)

$med = median($len, @my_list);
print "The median of \@my_list is: $med \n";

9.9 Pattern Matching

- The pattern-matching operator is m, but if slashes are used to delimit the pattern operand, the m can be omitted
  - The default string against which the pattern is matched is in \$_
- Character and character-class patterns
    - A non-meta, or normal character matches itself if (/gold/) {
      print "There's gold in that thar string!! \n";
    }
    - Metacharacters can match themselves if they are backslashed
  - A non-meta, or normal character matches itself
  - The period matches any character except newline
    /a.b/ matches "a\b", "a\bh", "ab", ...
9.9 Pattern Matching (continued)

- A character class is a string in brackets
  \[abc\] means a | b | c
- A dash can be used to specify a range of characters
  \[A-Za-z\]
- If a character class begins with a circumflex, it means the opposite
  \[^A-Z\]

- Predefined character classes:

<table>
<thead>
<tr>
<th>Name</th>
<th>Equivalent Pattern</th>
<th>Matches</th>
</tr>
</thead>
<tbody>
<tr>
<td>\d</td>
<td>[0-9]</td>
<td>a digit</td>
</tr>
<tr>
<td>\D</td>
<td>[^0-9]</td>
<td>not a digit</td>
</tr>
<tr>
<td>\w</td>
<td>[A-Za-z_0-9]</td>
<td>a word character</td>
</tr>
<tr>
<td>\W</td>
<td>[^A-Za-z_0-9]</td>
<td>not a word character</td>
</tr>
<tr>
<td>\s</td>
<td>[ \r\t\n\f]</td>
<td>a whitespace character</td>
</tr>
<tr>
<td>\S</td>
<td>[^ \r\t\n\f]</td>
<td>not a whitespace character</td>
</tr>
</tbody>
</table>

9.9 Pattern Matching (continued)

- Pattern Quantifiers

- pattern \{n\} means repeat the pattern n times
  /\d\{5\}/
- pattern* means repeat the pattern zero or more times
  /\d*/
- pattern+ means repeat the pattern 1 or more times
  /\d+/ 
- pattern? means zero or one match
  /\d*b?c+/ 

- Two more useful predefined patterns:
  \b - matches the boundary position between a \w character and a \W character, in either order
  \B - matches a non-word boundary
  - These two do not match characters, they match positions between characters

9.9 Pattern Matching (continued)

- Remembering matches

- After the match, the implicit variables $1, $2, ..., have the parts of the string that matched the first, second, ... parenthesized subpattern
  "John Fitzgerald Kennedy" =~ /\w\s\w\s\w/;
  Now, $1 has "John", $2 has "Fitzgerald", and $3 has "Kennedy"
- Inside the pattern, \s, \w, ... can be used
  $ has part of the string before the part that matched
  \s has part of the string after the part that matched
- Substitutions
  - Used to find and replace a substring
  $x/Pattern/New_String/
  $x = "Darcy is her name, yes, it's Darcy"
  $x/Darcy/Darcie/;

9.9 Pattern Matching (continued)

- Substitutions (continued)

- Modifiers

- The m modifier means find and replace all of them in the string
  The m modifier means the New_String must be interpreted as Perl code
  Example: Find a single hex character and replace it with its decimal value
  s/([\DA-Fa-f])/pack("C", hex($1))/m;
- The n modifier does what it does for pattern matching
- Transliterate Operator
  Translates a character or character class into another character or character class
  tx/a-z/A-Z/;
  Transliterates all lowercase letters to upper case

9.10 File Input and Output

- The open function is used to create the connection between a filehandle and the external name of a file; it also specifies the file's use

- A file's use is specified by attaching < (input), > (output, starting at the beginning of the file), or >> (output, starting at the end of the existing file) to the beginning of its name
  open (INDAT, "<prices");
  open (OUTDAT, ">>salaries");
  - Because open can fail, it is usually used with die open (OUTDAT, ">averages") or die "Error - unable to open salaries $!";
- One line of output to a file:
  print OUTDAT "The answer is: $result 
";
- One line of input from a file:
  $next = <INDAT>;
- Buffers (of any size) of input can be read from a file with the read function
  $result = <INDAT>;