Perl Programming

Subroutines

- Subs group related statements into a single task
- Perl allows both declared and anonymous subs
- Perl allows various ways of handling arguments
- Perl allows various ways of calling subs
- `perldoc perlsub` gives complete description

Previous http://stuff.mit.edu/iap/perl/ Next

Perl Programming

Declaring Subroutines

- Subroutines are declared with the `sub` keyword
- Subroutines return values
  - Explicitly with the `return` command
  - Implicitly as the value of the last executed statement
- Return values can be a scalar or a flat list
  - `wantarray` describes what context was used
  - Unused values are just lost

```
sub ten {
    return wantarray() ? (1 .. 10) : 10;
}

@ten = ten();           # (1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
```

```
$ten = ten();          # 10
($ten) = ten();        # (1)
($one, $two) = ten();  # (1, 2)
```

Previous http://stuff.mit.edu/iap/perl/ Next

Perl Programming

Handling Arguments

- Two common means of passing arguments to subs
  - Pass by value
  - Pass by reference
  - Perl allows either
- Arguments are passed into the @_ array
  - @_ is the "fill in the blanks" array
  - Usually should copy @_ into local variables

```
sub add_one {           # Like pass by value
    my ($n) = @_;       # Copy first argument
    return ($n + 1);    # Return 1 more than argument
}

sub plus_plus {         # Like pass by reference
    $_[0] = $_[0] + 1;  # Modify first argument
```

```
}

my ($a, $b) = (10, 0);

add_one($a);            # Return value is lost, nothing changes
$b = add_one($a);       # $a is 10, $b is 11

plus_plus($a);          # Return value lost, but a now is 11
$b = plus_plus($a);     # $a and $b are 12
```

Previous http://stuff.mit.edu/iap/perl/ Next

Perl Programming

Calling Subroutines

- Subroutine calls usually have arguments in parenthese
  - Parentheses are not needed if sub is declared first
  - But using parentheses is often good style
- Subroutine calls may be recursive
- Subroutines are another data type
  - Name may be preceded by an & character
  - & is not needed when calling subs

```
print factorial(5) . "\n";   # Parentheses required

sub factorial {
    my ($n) = @_;
    return $n if $n <= 2;
    $n * factorial($n - 1);
}
```

```
print ((factorial 5) . "\n");    # Parentheses around argument not required,
                                 # but need to ensure there are no extra argume
print &factorial(5) . "\n";  # Neither () nor & required
```

Previous http://stuff.mit.edu/iap/perl/ Next

Perl Programming

Subroutines - Example

- A more typical example
  - Declare subroutine
  - Copy arguments
  - Check arguments
  - Perform computation
  - Return results

```
sub fibonacci {
    my ($n) = @_;
    die "Number must be positive" if $n <= 0;
    return 1 if $n <= 2;
    return (fibonacci($n-1) + fibonacci($n-2));
}

foreach my $i (1..5) {
    my $fib = fibonacci($i);
```

```
    print "fibonacci($i) is $fib\n";
}
```

```
fibonacci(1) is 1
fibonacci(2) is 1
fibonacci(3) is 2
fibonacci(4) is 3
fibonacci(5) is 5
```

Previous http://stuff.mit.edu/iap/perl/ Next

Perl Programming

References

- References indirectly refer to other data
- Dereferencing yields the data
- References allow you to create anonymous data
- References allow you to build hierarchical data structures

Previous http://stuff.mit.edu/iap/perl/ Next

Perl Programming

Referencing Data

- References indirectly refer to other data
  - References are like pointers, but done right
  - Backslash operator creates a reference
  - References are scalars

```
my @fruit = qw(apple banana cherry);
my $fruitref = \@fruit;
```

Previous http://stuff.mit.edu/iap/perl/ Next

Perl Programming

Dereferencing Data

- Dereferencing yields the data
  - Appropriate symbol dereferences original data
  - Arrow operator (->) dereferences items

```
my @fruit = qw(apple banana cherry);
my $fruitref = \@fruit;

print "I have these fruits: @$fruitref.\n";
print "I want a $fruitref->[1].\n";
```

```
I have these fruits: apple banana cherry.
I want a banana.
```

Previous http://stuff.mit.edu/iap/perl/ Next

Perl Programming

Anonymous Data

- References allow you to create anonymous data
  - Referencing arrays and hashes are common
  - Build unnamed arrays with brackets ([ ])
  - Build unnamed hashes with braces ({ })

```
my $fruits = ["apple", "bananas", "cherries"];
my $wheels = {unicycle => 1,
              bike     => 2,
              tricycle => 3,
              car      => 4,
              semi     => 18};

print "A car has $wheels->{car} wheels.\n";
```

```
A car has 4 wheels.
```

Previous http://stuff.mit.edu/iap/perl/ Next

Perl Programming

Hierarchical Data

- References allow you to build hierarchical data structures
  - Arrays and hashes can only contain scalars
  - But a reference is a scalar, even if it refers to an array or a hash
  - Don't even need the arrow operator for these structures

```
my $fruits  = ["apple", "bananas", "cherries"];
my $veggies = ["spinach", "turnips"];
my $grains  = ["rice", "corn"];
my @shopping_list = ($fruits, $veggies, $grains);

print "I should remember to get $shopping_list[2]->[1].\n";
print "I should remember to get $shopping_list[0][2].\n";
```

```
I should remember to get corn.
I should remember to get cherries.
```

Previous http://stuff.mit.edu/iap/perl/ Next

Perl Programming

Files

- Access to files is similar to shell redirection
- Standard files
- Reading from files
- Writing to files
- Pipes
- File checks

Previous http://stuff.mit.edu/iap/perl/ Next

Perl Programming

File Access

- Access to files is similar to shell redirection
  - `open` allows access to the file
  - Redirect characters (`<, >`) define access type
  - Can read, write, append, read & write, etc.
  - Filehandle refers to opened file
  - `close` stops access to the file
  - `$!` contains IO error messages
  - `perldoc perlopentut` has complete description

```
open INPUT,  "<  datafile" or die "Can't open input file: $!";
open OUTPUT, ">  outfile " or die "Can't open output file: $!";
open LOG,    ">> logfile " or die "Can't open log file: $!";
open RWFILE, "+< myfile  " or die "Can't open file: $!";
```

```
close INPUT;
```

Previous http://stuff.mit.edu/iap/perl/ Next

Perl Programming

Standard Files

- Standard files are opened automatically
  - `STDIN` is standard input
  - `STDOUT` is standard output
  - `STDERR` is standard error output
  - Can re-open these for special handling
  - `print` uses standard output by default
  - `die` and `warn` use standard error by default

```
print STDOUT "Hello, world.\n";  # STDOUT not needed

open STDERR, ">> logfile" or die "Can't redirect errors to log: $!";
print STDERR "Oh no, here's an error message.\n";
warn "Oh no, here's another error message.\n";
close STDERR;
```

Previous http://stuff.mit.edu/iap/perl/ Next

Perl Programming

Reading from Files

- Reading from files
  - Input operator <> reads one line from the file, including the newline character
  - chomp will remove newline if you want
  - Can modify input recorder separator $/ to read characters, words, paragraphs, records, etc.

```
print "What type of pet do you have? ";
my $pet = <STDIN>;                      # Read a line from STDIN
chomp $pet;                             # Remove newline

print "Enter your pet's name: ";
my $name = <>;                          # STDIN is optional
chomp $name;

print "Your pet $pet is named $name.\n";
```

```
What type of pet do you have? parrot
Enter your pet's name: Polly
Your pet parrot is named Polly.
```

Previous http://stuff.mit.edu/iap/perl/ Next

Perl Programming

Reading from Files

- Reading from files
  - Easy to loop over entire file
  - Loops will assign to $_ by default
  - Be sure that the file is open for reading first

```
open CUSTOMERS, "< mailing_list" or die "Can't open input file: $!";

while (my $line = <CUSTOMERS>) {
  my @fields = split(":", $line);      # Fields separated by colons
  print "$fields[1] $fields[0]\n";     # Display selected fields
  print "$fields[3], $fields[4]\n";
  print "$fields[5], $fields[6]  $fields[7]\n";
}

print while <>;                        # cat
print STDOUT $_ while ($_ = <STDIN>);  # same, but more verbose
```

```
Last name:First name:Age:Address:Apartment:City:State:ZIP
Smith:Al:18:123 Apple St.:Apt. #1:Cambridge:MA:02139

Al Smith
123 Apple St., Apt. #1
Cambridge, MA  02139
```

Previous http://stuff.mit.edu/iap/perl/ Next

Perl Programming

Writing to Files

- Writing to files
  - `print` writes to a file
  - `print` writes to a STDOUT by default
  - Be sure that the file is open for writing first

```
open CUSTOMERS, "< mailing_list" or die "Can't open input file: $!";
open LABELS,    "> labels"       or die "Can't open output file: $!";

while (my $line = <CUSTOMERS>) {
  my @fields = split(":", $line);
  print LABELS "$fields[1] $fields[0]\n";
  print LABELS "$fields[3], $fields[4]\n";
  print LABELS "$fields[5], $fields[6]  $fields[7]\n";
}
```

Previous http://stuff.mit.edu/iap/perl/ Next

Perl Programming

Pipes

- Pipes
  - Pipes redirect input from or output to another process
  - Just like shell redirection, pipes act like normal files

```
# Use another process as input
open INPUT, "ps aux |" or die "Can't open input command: $!";

# Print labels to printer instead of to a file
open LABELS, "| lpr" or die "Can't open lpr command: $!";
```

Perl Programming

File Checks

- File checks
  - File test operators check if a file exists, is readable or writable, etc.
  - `-e` tests if file is exists
  - `-r` tests if file is readable
  - `-w` tests if file is writable
  - `-x` tests if file is executable
  - `-l` tests if file is a symlink
  - `-T` tests if file is a text file
  - `perldoc perlfunc` lists more

```
my $filename = "pie_recipe";
if (-r $filename) {
```

```
    open INPUT, "> $filename" or die "Can't open $filename: $!";
} else {
    print "The file $filename is not readable.\n";
}
```

Previous http://stuff.mit.edu/iap/perl/ Next