Modularity and Reusability I

Functions and code reuse

On being efficient

- When you realize that a piece of Perl code that you wrote may be useful in future programs, you may wonder if there is an easy way to reuse it, other than copying and pasting it into these programs (provided that you can even remember where you put it.)
- Even in a single program, you may find that there is some sequence of instructions that has to get repeated in several places in a program and may wonder whether there is a more efficient way of repeating them other than copying them over and over.

Example

- As an example, suppose that a program that you are writing contains several arrays, and each must be printed in the same format, with array elements one per line with numbered lines. You would need something like this in many places in the program:

```
$count = 0;
foreach $item ( @the_given_array ) {
    printf "%d  %s\n",$count++, $item;
}
```

- Wouldn't it be useful if there were a way to give this code a name, like a ***mini-program***, and just call the program whenever you need to print an array?

Subprograms

- You cannot simply put this code into a separate program because one program cannot call another directly.
- More importantly, even if you could, your program would have no way of giving the other program the array data in order to print it.
- What you really need is something like a program contained in your program that can be called from within your own program.
- Such things exist. They used to be called, quite naturally, ***subprograms***, but that term has gone by the wayside. Now subprograms are known as ***functions***.

Not such a far-fetched idea

- The idea of having a subprogram inside a program dates back to the first high-level language, Fortran. In Fortran, subprograms were called ***subroutines***.
- Whether you realized it or not, you have used subprograms in your Perl programs. Things like **`print`**, **`rand`**, **`chomp`**, **`push`**, **`split`**, and **`int`** are subprograms. We were calling them functions.
- In the early days, people distinguished two different kinds of subprograms: those that performed an action but did not "return" a value, and those that "returned a value."

Return values

- We tend not to use the value that **`print`**, **`push`**, or **`chomp`** returns; their usefulness is that they perform an action, not that they return something. We write things like

```
print @codons;
```

because we want **`print`** to do something for us, not to return a value.

- On the other hand, the ***only*** reason we use **`rand`**, **`split`**, and **`int`** is for the value they return. For example, we write

```
$number = rand(10);
```

so that **`rand`** can return a random number that we then store into **`$number`**.

Functions

- In some languages, there is a distinction between subprograms that return values and those that do not. ***In Perl, all subprograms return a value, whether or not we use it***, and ***subprograms are called functions***.
- The functions that are "part of" Perl, such as **`print`**, **`rand`**, and **`chomp`**, are called ***pre-defined functions***.
- We will spend a little time understanding these before writing our own.

Pre-defined functions

- You have seen so far that a function, such as **`print`**, is a piece of code that a program can call, possibly passing parameters to it. That piece of code is executed, and when it finishes execution, the program that called it continues its execution at the statement immediately after the call to the function.
- The next slide illustrates how execution flows through a function call.

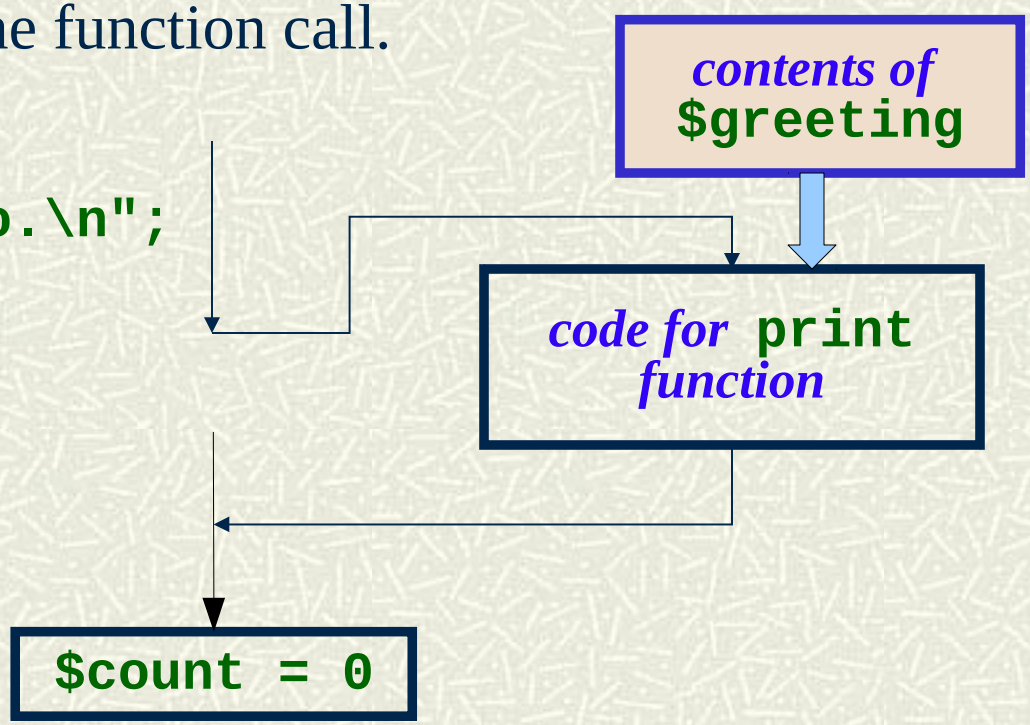
Function control flow

When a function is called, the flow jumps to the code of the function, flows through that code, and returns to the point immediately following the function call.

```
my $greeting = "Hello.\n";

print $greeting;

my $count = 0;
# more stuff follows
```



Data passed to a function

- The **`print`** function is supplied with a list of values to be printed. In Perl, functions such as **`print`**, **`rand`**, and **`chomp`** have one or more arguments that are provided following the function name, or enclosed within parentheses, e.g.:

  `rand(10)` or **`chomp($string)`**

- The functions themselves have a way to extract the values of these arguments. We will soon see how you can write functions that can also be passed argument lists.
- To start we look at functions without any arguments.

User-defined functions

- Perl lets users define their own functions, which are called ***user-defined functions***. For simplicity, most people just call them ***functions*** when the meaning is clear.
- To ***define*** your own function, you use the syntax

  sub ***function-name*** **{** ***block*** **}**

  in which ***function-name*** is a name you choose for the function and ***block*** is a sequence of zero or more statements.
- **sub** is a Perl keyword that introduces ***function definitions***. (It comes from Basic, which called them ***subroutines***; note the reference to their origin.)

Function example 1

- The following is a simple function definition.

```
sub DisplayGreeting
{
    print "Welcome to the program.\n";
}
```

- This both ***declares*** and ***defines*** a function that prints a welcome message on the screen. Later I will explain the difference between declaring a function and defining it.
- This function does something and returns no value.

Function example 2

- The following is a function that ***returns a value*** and does nothing else.

```
sub pi
{
    return 4*atan2(1,1);
}
```

- The simplest way to return a value is to use the **`return`** statement:

  `return` ***`expression`***

  evaluates and returns the expression in either scalar or list context, depending on how it is used in the calling program, and then terminates the function.

Naming functions

- Function names can be any legal Perl identifiers. Function names have their own namespace, like scalars, arrays, and hashes. This means that you can have a function **`foo`**, a scalar **`$foo`**, an array **`@foo`**, and a hash **`%foo`**.

- If a function is used primarily to do something, you should name it with a verb, such as **`DisplayGreeting`**, or **`ConvertString`**, or **`CleanUp`**. Names like these convey to the reader that the function is taking some specific action.

Naming functions (2)

- If a function is used primarily to return a value, you should name it with a noun, such as

  `DistanceBetweenPoints`, or

  `ShortestPath`, or

  `UsersInput`.

- Names like these convey to the reader that the function is returning specific data to the calling program.

Where do function definitions belong?

- Function definitions can go anywhere in a Perl program, but that does not mean that you should put them anywhere.
- All of the functions that you define should be placed either
  - in the beginning of the program, after all `use` pragmas, or
  - after the "main" program, i.e., at the end of the file, or
  - in a separate file, which would be included in the program with a `use` pragma.
- ***For small programs, put all function definitions at the beginning or the end of the program.***

An example program

- The following is a small program with a function.

```
#!/usr/bin/perl -w
use strict;
# Define function DisplayGreeting:
sub DisplayGreeting
{
    print "Welcome to the program.\n";
}
#*****************************************
# main program starts here:
DisplayGreeting();  # Call the function
```

Calling functions

- The preceding program demonstrates that one way to call a program is to write the program name followed by a pair of empty parentheses:

```
DisplayGreeting();
```

- If the function has arguments, then these would be put in a comma-separated list between the parentheses. We will get to that later.
- An alternative to using the parentheses is to prepend an ampersand to the name:

```
&DisplayGreeting;
```


Functions with arguments

- Arguments are passed to user-defined functions in the same way that they are passed to predefined functions, either
  - within the parentheses in a comma-separated list, or
  - without parentheses, as a list following the function (but only if the function name begins with "`&`" or its definition appears before the call.
- The arguments to a function are placed into the special array variable `@_`, which is available inside the function. The next slide demonstrates.

A function with one argument

- Notice below that **$_[0]** is the first element of @_; the value in **$name** is copied into it during the call.

```
sub DisplayGreeting
{   # $_[0] is first argument to DisplayGreeting
    print "$_[0], Welcome to the program.\n";
}
#****************************************
# main program starts here:
print "Enter your name:";
chomp( my $entered_name = <STDIN> );
DisplayGreeting($entered_name);
```

More about function arguments

- The elements of the `@_` array are **`$_[0]`**, **`$_[1]`**, and so on, *NOT* **`$[0], $[1]`**! ***Remember the underscore !!***
- It is a bad idea to use these variables inside the function because the names are not meaningful. It is better to immediately copy them into variables declared within the function, as in this version of **`DisplayGreeting`**:

```
sub DisplayGreeting
{
    my $name = $_[0];
    print "$name, Welcome to the program.\n";
}
```

About the @_ array

- The elements of the @_ array act as ***aliases*** for the list of arguments passed to the function.
- Remember from the description of the **`foreach`** statement that an ***alias is just another name for a variable***.
- The fact that @_ is an array of aliases means that any changes made within the function to **`$_[0]`**, **`$_[1]`**, etc. are actually made to the arguments in the calling program, as the next slide will demonstrate.
- Some terminology first: the variables **`$_[0]`**, and so on are called ***parameters*** of the function. The values that are passed to the function are called ***arguments***.

Example of aliasing effect

- The following program calls the **`double()`** function, defined below, which doubles the array elements' values.

```
#************ main program  ***************
 my @values = ( 1, 2, 4, 8 );
 double( @values );
 print "@values\n";  # prints 2 4 8 16

 # function definition:
 sub double {
     foreach my $i ( @_ ) {
         $i = $i * 2;
     }
 }
```

Preventing the aliasing effect

- Sometimes the change caused by the aliasing effect is what you want; sometimes, not.
- If you need to prevent the changes made in a function from changing the actual arguments in the caller, you should first copy the values from the `@_` array into lexical variables, as the next slide demonstrates.
- A lexical variable, by the way, is a variable defined using the **`my`** keyword. This concept will be covered later.

Preventing the aliasing effect: example

```
#************ main program  ***************
my @values = ( 1, 2, 4, 8 );
printDouble( @values );

# function definition:
 sub printDouble {
     my @list = @_; # copy into lexical
     foreach my $i ( @list ) {
         $i *= 2;
     }
     print "@list\n";
 }
```

More about function return values

- Remember that

  ```
  return <expression>;
  ```

  does two things: (1) it terminates the function and (2) it supplies the value of the expression as the return value of the function.
- If you do not put a `return()` statement in your function, the ***function will return the value of the last expression evaluated during its execution.***
- Even if a function does not return a value, ***the*** `return` ***statement is a way to stop a function from continuing.***

Returning a value without `return()`

- This is a variation of the `pi` function that we saw earlier, without the return statement. It returns the value of pi because the last expression evaluated has the value of pi.
- It also has a meaningless statement that has no effect on anything, just to demonstrate that only the last expression evaluated matters.

```
sub pi
{
    my $number = 10;
    4*atan2(1,1);
}
```

Using **`return`** to exit the function

- This function has a **`return`** in the **`while`** block, even though it does not return a value. It prints the lines from the input file until it finds the first line that matches the string passed to it in **`$_[0]`**. It uses **`return`** to exit immediately.

```
sub match {
    my ($line, $key) = ("", $_[0]);
    while ( chomp($line = <>) ) {
        if ( $line eq $key ) {
            return;
        } else {
            print "$line\n";
        }
    }
}
```

Return value context

- Functions can return scalars or lists. The following returns a list, for example:

```
sub randlist {
    my ($size, $limit) = @_;
    my @list = ();
    for (my $i = 0; $i < $size; $i++) {
        push @list, int(rand($limit));
    }
    return @list;
}
```

- This function creates a list of **`$size`** many random integers in the range [**`0..$limit-1`**].

Determining context

- The return value of a function will be evaluated in whatever context the function is called. If I called the preceding function in a scalar context, it would evaluate to the size of the list.
- You can design a function to check what context it is called in and return the appropriate value for each context. The `wantarray()` function returns true if the function is called in a list context and false if it is called in a scalar context. By checking the `wantarray()` function within your subroutine, you can select the return value. The next slide demonstrates this.

`wantarray()` example

- In scalar context, this function returns the concatenation of all strings in the array argument passed to it, but in array context, it returns the array itself, unmodified.

```
sub all {
    if ( !wantarray() ) {
        return join( "",@_ );
    }
    else {
        return @_;
    }
}
```

Passing multiple list arguments

- When you pass several arguments to a function, they are all stored in consecutive positions in the @_ array. In other words, they are flattened out in the array. For example, in

```
my @words = qw(words of silent prayer. );
my @jabberwocky = qw(The slithy toves wabe );
Upcase( @words, @jabberwocky );
print "@words\n@jabberwocky\n";
sub Upcase {
    foreach ( @_ ) { tr/a-z/A-Z/; }
}
```

- the @_ array in **Upcase** sees a single array.

Passing multiple list arguments (2)

- The problem with this is that the function does not "know" where one array ends and the other starts. In fact, it does not know how many arguments are passed to it. The only information it has is the total number of words and their values.
- But what if the function needs to know? For example, what if we wanted a function that created a times table with the elements in two numerical arrays. The next slide demonstrates the problem.

A Times Table

- We want, given (1, 2, 3, 4) and (1, 2, 3, 4, 5), to generate the table

*	1	2	3	4	5
1	1	2	3	4	5
2	2	4	6	8	10
3	3	6	9	12	15
4	4	8	12	16	20

- More generally, we would like a function, that given any arrays `@x` and `@y`, creates the table whose ***ij*th** entry is $x_i * y_j$.

An incorrect solution

- If we tried the following function, it would not work:

```
sub makeTable {
    my (@row, @col ) = @_;
    foreach my $r ( @row ) {
        foreach my $c ( @col ) {
            print $r*$c, "\t";
        }
        print "\n";
    }
}
makeTable(@x, @y);
```

- The array **@row** would contain all of **@x** and **@col** would be empty. Passing lengths of each would work, but it is messy.

References solve the problem

- The easiest solution is to pass ***references*** to the arrays instead of the arrays themselves, and within the function, to ***dereference*** the argument list, as is now shown.
- In the main program, if the row size and column size are typed on the command line, then this would be how to call the function:

```
my @rows = (1..$ARGV[0]);
my @columns = (1..$ARGV[1]);
makeTable(\@rows, \@columns);
```

- Notice that **`makeTable`** is passed references to the arrays, not the arrays themselves.

Using references (2)

- The **`makeTable()`** function is thus:

```
sub makeTable {
    my ($rowref, $colref ) = @_;
    foreach my $r ( @$rowref ) {
        foreach my $c ( @$colref ) {
            print $r*$c, "\t";
        }
        print "\n";
    }
}
```

- Notice that the function expects to receive references, and therefore it provides **`$rowref`** and **`$colref`** as references and ***dereferences*** them in the loops.

It could have been simpler, but…

- In its simplest form, we could have just passed in the row size and column size and let **`makeTable()`** write the table without even using arrays. We didn't really need the function for that.
- However, what we just wrote lets us form the pairwise products of arbitrary arrays such as

```
@x = (3, 5, 11, 2);
@y = (4, 9, 12, 15, 23 );
makeTable(\@x, \@y);
```

  which computes the pairwise products of the two vectors shown in the next slide.

An all-pairs products table

- The all-pairs products of the two vectors (3, 5, 11, 2) and (4, 9, 12, 15, 23):

*	4	9	12	15	23
3	12	27	36	45	69
5	20	45	60	75	115
11	44	99	132	165	253
2	8	18	24	30	46

Function libraries

- Once you have written a collection of functions that you think you might reuse in other programs, you may wonder where you should keep them so that you can use them later.
- In effect, what you would need is like a library, but instead of its being filled with books, it is filled with functions.
- Soon you will see how to create ***modules***, which are files containing functions and data that can be imported into your programs,

Reasons to use functions

- When the same or similar code repeats many times in a program, converting it into a function makes the program smaller and easier to modify and debug. (If you have to make the same change in 5 places, it is easy to make a mistake.)
- Even if the code does not repeat, when a chunk of a program is performing some single task, it is better to make it a function to consolidate its logic and give it a name. By giving it a name, you add to the set of clues about what it does.
- By making chunks of code functions, the main program gets smaller and starts to read like a list of small tasks, making it easier to understand and less intimidating to read.