A First Perl Project

Building a Program from Scratch

Overview

- In this lesson we develop a Perl program from scratch in order to demonstrate the ***process*** of program development.
- The problem statement will be initially vague, as they usually are. ***Problem refinement*** is an iterative process in which, as the user or client interacts with you, the developer, he or she clarifies the objectives and the details of the program. The ultimate goal of this process is to create a precise and unambiguous problem statement, called a ***requirements specification***, in software engineering jargon.
- In this scenario, you are the developer and the client is imaginary.

The problem

- We would like a program that repeatedly displays simple multiplication and division problems, waits for the user's answer, and checks the answer for correctness.
- Each multiplication problem will have single-digit operands.
- Each division problem shall be the inverse of a multiplication problem with single-digit operands, so that the answer will also be a single digit.
- The program will display a prompt character '**>**' to let the user know it is waiting for an answer.
- The program will terminate when the user presses the '**q**' key.

Examples of problems

- These are the kinds of problems it should create:

```
8  x 9 = ?
63 / 7 = ?
56 / 7 = ?
5  x 0 = ?
```

- It can use the '**x**' to represent multiplication and '**/**' to represent division.

Example session

- A session may look like

```
$ mathq
Welcome to the math question program.
4 x 8 = ?
> 32
Correct!
54 / 6 = ?
> 8
Incorrect. 54 / 6 = 9
7 * 7 = ?
> q
Thank you for playing mathq. Bye.
$
```

Handling user input

- There are four possible responses from the user:
  - He/she typed a correct numerical answer
  - He/she typed an incorrect numerical answer
  - He/she typed **q** to quit.
  - He/she typed something "invalid".
- The Problem Statement is supposed to tell us what the program should do in the last case. If it does not, it is up to use to decide what is reasonable. Most likely our choice will not make the client happy, so that will result in a change in the Problem Statement.

Handling bad user input

- For this first project, we, the developers, shall make the reasonable decision that if the user enters any input other than a number or the letter 'q', the program will display some simple error message and redisplay the question and prompt.
- The error message will be

```
"That was an invalid response. Enter an integer or
'q' to quit."
```

Program design

- Having created an informal but precise problem statement, we are ready to design the program.
- There are several schools of thought about how to design a program. ***We will start with a top-down approach*.**
- In this approach, we start with descriptions of the sweeping tasks that the program needs to accomplish, and we gradually take each task in turn and decompose it into smaller tasks.
- Eventually the small tasks get converted to program code.

First pass

- The following are some of the tasks the program needs to do

```
display the startup message
generate a random math question and solution
display the question and get valid response
check correctness of user's response
display the exit message
```

- This is not a description of the program flow; it is just a set of tasks that the program has to do.

Second pass

- We now write a rough design for the program, using the tasks we just described.

```
display the startup message
LOOP: until the user wants to quit:
{
    generate a random math question and solution
    display the question and get valid response
    check correctness of user's response
} end of LOOP
display the exit message
```

Pseudo-code

- The previous description of the program is written in a language commonly called ***pseudo-code***. Pseudo-code is like a ***pidgin*** language of computer scientists; it is a language that is part programming language and part English, evolved to describe programs in an English-like way.
- The textbook by Andrew Johnson uses a style of pseudo-code based on an idea called Literate Programming, invented by Don Knuth. We will follow it only loosely.
- Each pseudo-code task is enclosed in **<< >>** pairs and is called a chunk.
- If I use a Perl keyword anywhere, I will underline it.

Literate programming version

```
<<mathq>>==
display the startup message
LOOP: until the user wants to quit:
{
    <<generate a random math question and solution>>
    <<display the question and get valid response>>
    <<check correctness of user's response>>
} end of LOOP
display the exit message
```

- Some tasks are not complex enough to be considered chunks; they will be replaced by a single statement eventually.

Chunks

- The first line,

  `<<mathq>>==`
- is a chunk definition. It means that the the task `<<mathq>>` is defined by the following sequence of statements.
- `mathq` is the outermost chunk. We now take each chunk in turn and decompose it into programming statements. We start with generating questions and solutions.

Generating math questions

- We need to pick two one-digit numbers randomly. We also need to pick an operation, multiplication or division, randomly.
- Fortunately, Perl has a ***function*** that can generate random numbers, so we can use that function to solve this problem.
- A ***function*** in a programming language is like a mathematical function. It has arguments and it has a value. For example, in math, if we write $f(x) = x^2$

  then we say that $f(x)$ is a function whose value is x^2.
- In computer science we would say that $f(x)$ is ***a function that returns*** the square of x.

Generating math questions (2)

```
Task: generate a random math question and solution
    set first_num to a random integer in 0..9
    set second_num to a random integer in 0..9
    pick a random integer from 0 to 1
    if it is 1 {
        <<create mult question and solution>>
    } else {
        <<create division question and solution>>
    } #endif
```

- It is assumed that the questions will use the two numbers chosen in the first two steps.

The if-else statement

- The preceding chunk used the ***if* … *else*** structure that you saw briefly in an earlier lesson. Its syntax in Perl is

```
if ( expression ) {
        true_statement_block
} else {
        false_statement_block
} #endif
```

- If the expression has a true value, the ***true statement block*** will be executed. Otherwise the ***false statement bloc***k will be executed. After either, the next instruction to be executed is the one after the ***#endif*** comment. Look at the demo program in Chapter03 on our UNIX host.

Refining further: Multiplication question

- We now refine the two chunks inside the previous chunk.

```
Task: create mult question and solution
set solution to first_num times second_num
set question to "first_num x second_num = ?"
```

- In the question, we are not writing the literal words "***first_num***" or "***second_num***"; we are writing the numbers that were randomly generated. For example, if first_num is 7 and second_num is 8, the question would be

```
"7 x 8 = ?"
```

Refining further: Division question

- The division question requires some ingenuity. We want the answer to be a whole number. Therefore, we can generate a multiplication problem such as 7 x 6 = ?, calculate the answer, 42, and then swap the answer with the first number, turning into division at the same time: 42 / 6 = ?
- This will always give us an integer solution.

```
Task: create division question and solution
  set solution to first_num times second_num
  swap values of first_num and solution
  set question to "first_num / second_num = ?"
```

Incorporating refinements

- We can incorporate the preceding refinements into a single piece of code that will eventually become the instructions to implement the **`<<generate a random math question and solution>>`** chunk. That code is found in the **`perldemos/chapter03`** directory in the cs132 directory tree.
- We can repeat this process for each of the second level chunks written as tasks in **`<<mathq>>`**.

Refining the next chunk

- Displaying a question is easy, and getting the user's reply is also easy. But ensuring that it is a valid response requires looking at the response and, if it is not valid, making them try again, and doing this until they enter a valid response. This is, in effect, a loop:

```
Task: display question and get valid response
     set response_is_valid to FALSE
     LOOP: until response_is_valid is TRUE {
         display question
         get user's response
         <<check if response is valid>>
         #sets response_is_valid if it really is
     } end of LOOP
```

Notes

- The pseudo-code statement

  ```
  set response_is_valid to FALSE
  ```

  is almost like a Perl statement. It is almost explicitly stating that we need a variable, here named **`response_is_valid`**, and that we will set its value to the logical value **`FALSE`**. (Hopefully you know enough logic to know that there are just two values, **`FALSE`** and **`TRUE`**.)
- The hardest task is figuring out how to design the ***`<<check if response is valid>>`*** chunk.

Checking validity of input

- A task in every interactive programming project is validating input. It sometimes requires more code than the rest of the program.
- In this program, the task is not too difficult; the user is allowed to enter either all digits or the letter '**q**'. Therefore, this task is reduced to:

```
Task: check if response is valid
     if response is all digits or 'q' {
          set response_is_valid to TRUE
     } end of if
```

Checking correctness of user input

- The check for correctness is to compare the number that the user entered to the solution generated earlier:

```
<<check correctness of user's response>>==
    if response equals solution {
        display correct_response_message
    } else {
        display incorrect_response_message
    } end if
```

But the result of the chunk ***<<display question and get valid response>>*** could be '**q**' to quit. We need to test for that first.

Checking correctness, revised

- The complete chunk is therefore

```
Task: check correctness of user's response
  if response equals 'q' {
      set user_wants_to_quit to TRUE
  } else {
  if response equals solution {
      display correct_response_message
  } else {
      display incorrect_response_message
  } end if
```

Assembling the chunks

- All of the chunks have been written. The entire pseudo-code program can be assembled now. Take a look at the complete program, called **`mathq-pseudo-code.pl`**, in the **`Chapter03`** demo program directory.
- It is simply a matter of replacing the chunk descriptors by the chunk definitions that we wrote.
- Each of the lines in this program starts with a # so that it is treated like a comment. I can compile this program as is. It will do nothing. This way I can slowly convert the code to Perl and check as I go along.

Creating the Perl program: Line 1

- The Perl program can be written by converting the pseudo-code file line by line into legal Perl code. To do this, you need to know Perl. Since you have not learned Perl yet, this will be an introduction to Perl by example.
- We start with the first line,

```
# display the startup message
print "Welcome to the mathq program.\n";
```

- You can see that all we need is the Perl `print` statement, which you have seen before.

Creating the Perl program: Outer loop

- The next step is to encode the logic of the main loop of the program. The main loop is the outer loop, the one that contains the three chunks.
- Perl has an ***until loop***. The until loop looks like:

```
until (condition) {
        statement_block
}
```

- As long as the `condition` is false, the statement block is executed, and the condition is retested. When the condition is false, the program continues to the statement after the loop.

Creating the Perl program: Outer loop

- We need to declare and initialize the variable to control the loop. In Perl, there is no special value ***FALSE*** or ***TRUE***. The value ***0*** is ***FALSE*** (and so is the empty string). Any non-zero value is ***TRUE***. (Perl borrowed this idea from C.)
- The main loop is therefore:

```
my $user_wants_to_quit = 0;
until ($user_wants_to_quit) {
     statement_block
}
```

- We will rewrite this in the next slide, with the inner chunks and the closing statement.

The main loop

- This is the main loop so far. The next step is converting the chunks to code.

```
print "Welcome to the mathq program.\n";
my $user_wants_to_quit = 0;
until ($user_wants_to_quit) {
    <<generate a random math question and solution>>
    <<display the question and get valid response>>
    <<check correctness of user's response>>
}
print "Exiting the mathq program.\n";
```

Perl functions: `rand()`

- Perl has a function named **`rand()`** that works as follows. If you give **`rand()`** an argument, say 10, as in

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

  then **`rand()`** will generate a random number, possibly with a fractional part, at least 0 but less than 10, and return it. In this case it would be assigned to the variable **`$x`**.

- When we write the names of functions, like **`rand()`**, we always put empty parentheses after the name so that the reader knows it is a function. When you use it in a program, you put a value between the parentheses.

Creating random integers

- Because **`rand()`** returns numbers with fractions, we need to "chop off" the fraction. The function **`int()`** chops the fraction off of its argument. For example,

  `int(3.5)` has the value **`3`**

- To create two random integers between 0 and 9 and assign them to the variables **`$first_num`** and **`$second_num`**, we therefore compose **`int()`** on **`rand()`** as in

```
my $first_num  = int(rand(10));
my $second_num = int(rand(10));
```

Choosing the operator randomly

- To choose the operator randomly, **`int(rand(2))`** will be used. This will return either 0 or 1. (Why?)
- We arbitrarily decide that if it is a 0, we will use division and if it is 1, we use multiplication:

```
my $operator = int(rand(2));
if ( $operator == 1 )
    <<create mult question and solution>>
else
    <<create division question and solution>>
```

Declaring variables

- Sometimes as you write a program you will discover that you need more variables. The variable declarations (those "my" declarations) should be placed before the place you use the variables. Exactly where to put them depends on concepts you have yet to learn.
- For now, we need to declare two more variables: **`$question`** and **`$solution`**. **`$question`** will contain the text of the question and **`$solution`** will contain the numerical answer that we calculate.
- The entire ***`<<generate a question and its solution>>`*** is on the next slide.

The Code to generate math questions

```
my $first_num  = int(rand(10));
my $second_num = int(rand(10));
my $question;
my $solution;
my $operator = int(rand(2));
if ( $operator == 1 ) {
    $solution = $first_num * $second_num;
    $question = "$first_num x $second_num = ?";
} else {
    $solution = $first_num * $second_num;
    ($solution, $first_num) = ($first_num, $solution);
    $question = "$first_num / $second_num = ?";
}
```

Explanation

- In the preceding slide, the statement,
  `($solution, $first_num) = ($first_num, $solution);`
  is an example of a ***list assignment***. We will learn about these later. In effect, it copies the value from **`$first_num`** into **`$solution`** and copies the old value of **`$solution`** into **`$firstnum`**.
- Also, you should notice that there are variables inside the strings, as in

```
"$first_num x $second_num = ?"
```

- In this case, Perl first evaluates the variables, and then puts the values into the string.

Code to process input

- The next chunk is elaborated now. We need a variable to store the user response and one to store the test if it is valid:

```
my $response;
my $response_is_valid;
```

- The chunk once again is:

```
Task: display question and get valid response
       set response_is_valid to FALSE
       LOOP: until response_is_valid is TRUE {
           display question
           get user's response
           <<check if response is valid>>
       } end of LOOP
```

Code to process input (2)

- Our first pass is:

```
my $response;
my $response_is_valid = 0; # set to FALSE
until ( $response_is_valid ) {
    print "$question\n";  # display question
    get user's response
    <<check if response is valid>>
}
```

- We now have to fill in the code to get the user's response and check it.

Getting user input

- Reading the keyboard input is accomplished with:

```
$response = <STDIN>;
```

- This statement causes everything the user types on the keyboard up to and including the first **`<ENTER>`** keypress, to be stored in the variable **`$response`**. To remove the **`<ENTER>`** key character from $response, Perl has a function called **`chomp()`**. **`chomp()`** removes that character:

```
$response = <STDIN>;
chomp($response);
```

Incorporating the input

- Putting it together, we now have:

```
my $response;
my $response_is_valid = 0; # set to FALSE
until ( $response_is_valid ) {
    print "$question\n> ";  # display question
    $response = <STDIN>;
    chomp($response);
    <<check if response is valid>>
}
```

- Next we fill in the chunk that validates input.

Input validation chunk

- We need to complete this chunk:

```
Task: check if response is valid
          if response is all digits or 'q' {
              set response_is_valid to TRUE
          } end of if
```

- How can we test if **`$response`** is all digits or equals the letter **`q`**? With Perl's matching operator and regular expressions.
- Perl has regular expressions similar to **`grep`**'s.

Perl's match operator

- We can check whether a variable, such as `$response`, contains a string that matches a pattern using the expression:

```
        $response =~ m/pattern/
```

- where ***`pattern`*** is a regular expression with mostly the same syntax as that of **`grep`**. The **`=~`** operator is Perl's ***binding operator***. The variable on the left-hand side of **`=~`** is searched for a match of the pattern on the right-hand side. If a match is found, a true value is returned, otherwise a false value is returned.
- The **`m//`** operator is the ***`match operator`***. (You do not need the '**`m`**' -- **`//`** by itself also works.

Perl's match operator

- For example, to check if **`$response`** matches a line that contains at least one digit and nothing but digits:

```
        $response =~ m/^[0-9]+$/
```

  or equivalently,

```
        $response =~ m/^\d+$/
```

- since **`\d`** is the pattern that matches any digit, and **`+`** is the "***1 or more occurrences***" operator. Do you remember that **`^`** and **`$`** are anchors to the beginning and end of a line in **`grep`**? In Perl they anchor to the beginning and end of a string.

Comparing strings

- To check if **`$response`** is equal to a particular string, we can use the string comparison operator, **`eq`**:

```
 $response eq 'q'
```

  This returns true if and only if the string stored in **`$response`** is exactly '**`q`**'. Putting this together, we have:

```
 if  ( $response =~ m/^[0-9]+$/ or $response eq 'q' ){
         $response_is_valid  = 1;
 }
 else {
         print "Invalid Input:Enter an integer ";
         print "or 'q' quit\n";
}
```

Finishing up

- The last chunk to convert is the chunk that compares the response to the solution and displays the appropriate message.

```
Task: check correctness of user's response
    if response equals solution {
        display correct_response_message
    } else {
        display incorrect_response_message
    } end if
```

- This is easily transformed into code, as shown next.

Checking correctness

Task: check correctness of user's response

```
    if ($response == $solution) {
        print "Correct!\n";
    } else {
        print "Incorrect: $question $solution\n";
    }
```

- This completes the code. Take a look at the final program in the Chapter03 directory.

Is the program correct?

- If you typed the program correctly, you should have no syntax errors. If not, you might find some. When you run the program, Perl will tell you the line number on which they occur. You can use any line editor to find that line and fix the mistake.
- If the program runs, it is not necessarily correct. You need to test it with many different types of inputs, and try to make it fail.
- This particular program is correct, except for one error: it will occasionally generate a division by zero, such as 6/0 = ?
- Fixing this requires some redesign.

What next?

- You just had an overview of the development of a Perl program from problem statement to solution, like building a house starting with the client's ideas for what it is supposed to look like. This gives you the big picture.
- The method was a top-down approach, starting with large ideas and making them more detailed and precise with each step.
- You don't have to know Perl to design chunks. but you do have to learn the bricks and mortar of Perl to write the program. That is what follows.