CIS 101 Study Guide
Fall 2012

Don Colton
Brigham Young University–Hawaii

December 3, 2012

This is a study guide for the CIS 101 class, Introduction to Programming, taught by Don Colton, Fall 2012.

It is a companion to the text book for the class, Introduction to Programming Using Perl and CGI, Third Edition, by Don Colton.

The text book is available here, in PDF form, free.

http://ipup.doncolton.com/

The text book provides explanations and understanding about the content of the course.

This study guide is focused directly on the grading of the course, as taught by Don Colton.

Contents

Chapter 1

How Points Are Earned

Your final grade is based on the number of points you earn. The exact details are in the syllabus. A summary is here.

Effort: About 50% of the points you can earn are for effort, even if you are unable to perform well after putting in the effort.

Effort includes Study Points (about 330), Daily Update points (about 50), and in-class Activity points (about 120).

Effort includes studying and doing certain in-class activities. For study we ask you to certify that you studied a certain amount of time. For in-class activities, we will demonstrate a certain programming technique and ask you to follow along. Normally this means that you are typing something that is currently projected on the screen at the front of the class room. We have you type it so it will pass through your mind at least once (grin), and so you can have the experience of solving the typing mistakes that are almost inevitable.

Performance: About 50% of the points you can earn are for performance, even if it just comes naturally to you with no effort.

Performance includes correctly answering questions on exams (420). It also includes doing a final project (80).

The remainder of this study guide looks at each of the performance points and gives you the information we think you might need to master each skill.

Chapter 2

Activities

Contents

My intention is that we will do in-class activities many times through the semester. We assume you are studying outside of class time, and that the text book that I provide contains enough background information to avoid lots of lecturing in class.

Through the semester, this chapter will be modified as new activities are assigned. This will give students a reliable place where they can find information about each task.

2.1 oXX: Online General Rules

Online tasks generally follow these rules. Exceptions and clarifications are provided for each task.

Label: Each online task has a grading label consisting of the letter "o" (for online) followed by (normally) two other characters that specify which online task it is.

Task: Create a web page (a) properly linked to the CIS 101 student projects page. It should include (b) your name. Other requirements vary by task.

How To Submit: Create a web page properly linked to the student projects page. I normally grade everyone's submissions at once.

Late Work: If you complete or improve your work so that regrading may be justified, tell me so via email. Put "CIS 101 (grading label)" at the start of your subject line.

Grading Rules (Rubric):

Grade 0: missing (a), or fails to run.

Grade 1: exists but is missing one or more requirements.

Grade 2: (normal) meets all requirements.

Grade 3: (rare) is particularly impressive.

Grades will be posted to the "CIS 101 Activities" grade book in the column specified by the grading label.

Late Work Deadine: Two weeks after the task was discussed in class.

2.1.1 oWP: Web Page (online)

Grading Label: oWP

Task: Create a web page (a) properly linked to the student projects page. It should include (b) your name, (c) some bold text, (d) some italic text, and (e) a picture.

Late Work Deadine: Oct 31, 2012.

2.1.2 oML: Mad Lib (online)

Grading Label: oML

Task: Create a web page (a) properly linked to the student projects page. It should include (b) your name, (c) three or more input fields with suitable descriptions and (d) reasonable default values, (e) a submit button. When the submit button is pressed (e) the input values should not change, and (f) a story should be presented that uses the contents of the input fields.

Late Work Deadine: Oct 31, 2012.

2.1.3 oDI: Dice (online)

Grading Label: oDI

Task: Create a web page (a) properly linked to the student projects page. It should include (b) your name, (c) results of rolling two or more dice, both numerically and (d) pictorially.

Optional: Instead of representing dice, you can represent anything else that is suitable to this assignment, such as Rock Paper Scissors, or Heads Tails, or different Pokémon monsters.

Late Work Deadine: Oct 31, 2012.

2.1.4 oHL: High-Low (online)

Grading Label: oHL

Task: Create a web page (a) properly linked to the student projects page. It should include (b) your name, (c) a blank in which to place a guess, (d) a submit button, and (e) a hidden field with the number to be guessed. (f)

When the program first starts, it should pick a number to be guessed. (g) if the guess is too low, it should say so. (h) if the guess is too high, it should say so. (i) If the guess is correct, it should say so and (j) pick a new number to be guessed.

Late Work Deadine: Oct 31, 2012.

2.1.5 oNW: Now (online)

Grading Label: oNW

Task: Create a web page (a) properly linked to the student projects page. It should include (b) your name, (c) the current hours:minutes:seconds, (d) the current day, month, year.

Late Work Deadine: Nov 26, 2012.

2.1.6 oF1: Farm (online)

Grading Label: oF1

Task: Create a web page (a) properly linked to the student projects page. It should include (b) your name, (c) at least three blanks into which numbers can be typed, (d) and a submit button.

(e) Each number is associated with a "plant" for which that many plants will be grown. There should be a maximum number, probably 10, beyond which higher numbers do not result in more plants.

(f) Numbers should be "sticky" in the sense that when the screen updates, the number that was in each blank is still in that blank.

Recommendation: use a for loop for each kind of plant.

Late Work Deadine: Dec 3, 2012.

2.1.7 oF2: Farm (online)

Grading Label: oF2

Task: Same as oF1 (above) but using subroutines.

Make an array (list) of the plants you are farming.

```
@plants = ( ... );
```

For simplicity, the text strings in @plants must be usable for display labels, for input names, and for image file names.

Thus, if one of the plants is `"tomato"` you can display "tomato" as part of the display label, use `name="tomato"` in the input field, use `olin("tomato")` to retrieve the quantity, and use `"tomato.jpg"` to show the picture of the tomato.

Or, more specifically, if `$fruit = "tomato"` you can display `$fruit` as part of the display label, use `name="$fruit"` in the input field, use `olin($frult)` to retrieve the quantity, and use `"$fruit.jpg"` to show the picture of the tomato.

Use the following foreach loop to display the names and quantities of the plants.

```
foreach $plant ( @plants ) { ask ( $plant ) }
```

Then put an appropriate "submit button."

Use the following foreach loop to display the harvest.

```
foreach $plant ( @plants ) { show ( $plant ) }
```

Your task includes writing the `ask` and `show` subroutines. These subroutines will call olin as needed and do all the printing required to accomplish their tasks.

After you get it working, email it to me so I can verify you used the proper structure in writing your program.

Make "cis101 farm2 (yourname)" the subject line of your email, where (yourname) is replaced by your lastname, firstname.

Late Work Deadine: Dec 10, 2012.

2.2 cXX: Command-Line General Rules

Command-line tasks generally follow these rules. Exceptions and clarifications are provided for each task.

Label: Each command-line task has a grading label consisting of the letter "c" (for command-line) followed by (normally) two other characters that specify which online task it is.

Task: Write a program. Requirements vary by task.

How To Submit: Send your work to me via email. Put **CIS 101 (label)** at the start of your subject line, where (label) is replaced by the actual task label. Put your program as the body of the message. (Do not send it as an attachment.)

If your submission was not acceptable, I will reply to it giving at least one reason that it was not acceptable. (There may be other problems that I did not notice or mention.) Normally you should fix the problem and resubmit your program.

If your submission was acceptable, I will reply to it with the word "Success" and possibly additional comments.

Grading Rules (Rubric):

Grade 0: nothing acceptable was received.

Grade 1: (this grade is not used on these tasks.)

Grade 2: (normal) meets all requirements.

Grade 3: (rare) is particularly impressive.

Grades will be posted to the "CIS 101 Activities" grade book in the column specified by the grading label.

Late Work Deadine: Two weeks after the task was discussed in class.

2.2.1 cML: Mad Lib (command)

Grading Label: cML

Task: Write a program. Prompt for at least three inputs, such as "name of a boy" or "activity that is free". Then compose a story that uses those inputs.

Submission Deadine: Oct 31, 2012.

2.2.2 cHL: High-Low (command)

Grading Label: cHL

Task: Write a program. When it starts it should pick a number. It should then iterate (loop) asking for a guess, and telling whether it is too high, too low, or correct. After the correct answer is found, it should start over, picking another number. This will involve two infinite loops, nested.

Submission Deadine: Oct 31, 2012.

2.3 gXX: GradeBot Tasks

GradeBot itself is explained in 3 (page 15).

GradeBot tasks generally follow these rules. Exceptions and clarifications are provided for each task.

Label: Each GradeBot task has a grading label consisting of the letter "g" (for gradebot) followed by (normally) two other characters that specify which online task it is.

Lab ID: Each GradeBot task has a lab ID (name). Normally this ID consists of "cis101.(label).(word)" where (label) is the label above (probably without the g prefix), and (word) describes the lab.

Task: Write a program. GradeBot gives further directions for each program.

Have GradeBot test your program. When your program is running correctly, you will get this message:

```
Success!  No errors found.  Nice job.  Assignment complete!
```

After receiving this message, you can submit your program to me. I may require additional things, such as the use of specific program elements, or specific style.

How To Submit: Send your work to me via email. Put the **lab ID** at the start of your subject line.

Put your program as the body of the message. (Do not send it as an attachment.)

Include two comments at the top of each program: (1) Your name, (2) The lab ID.

If your submission was not acceptable, I will reply to it giving at least one reason that it was not acceptable. (There may be other problems that I did not notice or mention.) Normally you should fix the problem and resubmit your program.

If your submission was acceptable, I will reply to it with the word "Success" and possibly additional comments.

Grading Rules (Rubric):

Grade 0: nothing acceptable was received.

Grade 1: (this grade is not used on GradeBot tasks.)

Grade 2: (normal) meets all requirements.

Grade 3: (rare) is particularly impressive.

Grades will be posted to the "CIS 101 Activities" grade book in the column specified by the grading label.

Late Work Deadine: Two weeks after the task was discussed in class.

2.3.1 cis101.12.hi.Fred

Task: Ask for a name. Respond with "Hello, (name)!"

Late Work Deadline: Oct 31, 2012

2.3.2 cis101.21.phone

Task: Compare the cost of two phone plans for a planned phone call.

If the savings is less than 0.001 dollars, call them equal. Because of "round-off" problems, two numbers can be different even if they should be equal: 1/3=.333, (1/3)*3=.999.

You will need to format the savings to dollars and cents. In Perl, you can use one of these approaches to format it.

```
# printf means ‘‘print formatted’’
printf ( ‘‘blah blah %.2f blah blah\n’’, $savings );

# sprintf means ‘‘print formatted into a string’’
print ‘‘blah blah $savings blah blah\n’’;
```

Late Work Deadline: Oct 31, 2012

2.3.3 cis101.22.bday

Task: Ask for the user's name and birthday. If it is their birthday then wish them a happy birthday. Whether it is their birthday or not print out how old they are.

(You will be given today's date.)

Late Work Deadline: Oct 31, 2012

2.3.4 cis101.23.afford

Task: You are shopping for wedding gifts for a good friend. They have registered their wants one a bridal registry. There are two items not yet purchased. Ask for the price of gift 1. Ask for the price of gift 2. Ask for the amount of money you have. If you can get both, say so. If you can only get one, tell the most expensive thing you can afford. If you cannot afford either, say so.

Objective: Learn how to use if/else skillfully.

Late Work Deadline: Oct 31, 2012

2.3.5 cis101.31.while

Task: Read in three numbers: the starting point, the ending point, and the step size. Count from the starting point to the ending point, adding the step size each time, one number per line.

Use a normal "while" loop (pre-test, not interior test).

Note: if you do not hit the ending point exactly, that's okay. But do not go past the ending point.

Approved spacing and indenting is required for full credit.

Late Work Deadline: Oct 31, 2012

2.3.6 cis101.32.for

Task: Read in three numbers: the starting point, the ending point, and the step size. Count from the starting point to the ending point, adding the step size each time, one number per line.

Use a normal "for" loop (pre-test, not interior test).

Note: if you do not hit the ending point exactly, that's okay. But do not go past the ending point.

Approved spacing and indenting is required for full credit.

Late Work Deadline: Oct 31, 2012

2.3.7 cis101.33.last

Task: Read in three numbers: the starting point, the ending point, and the step size. Count from the starting point to the ending point, adding the step size each time, one number per line.

Use an infinite “while” loop (not a pre-test “while” loop) that uses an interior test to end the loop at the proper time.

Note: if you do not hit the ending point exactly, that’s okay. But do not go past the ending point.

Approved spacing and indenting is required for full credit.

Late Work Deadline: Oct 31, 2012

2.3.8 cis101.41.array

Start with an empty array named XYZ. Prompt for and read in a name. Add the name to the array. Repeat until you read in a blank name. See how many names are in the array. Report that number.

Approved spacing and indenting is required for full credit.

Late Work Deadline: Nov 5, 2012

2.3.9 cis101.42.roll

Task: Prompt for a student name. Add it to an array. Repeat until you read in a blank name. Tell how many students are attending.

Prompt again for the name of a student. Tell whether that student is present or absent.

Use push and foreach. Do not use indexing (like $x[1]). Approved spacing and indenting is required for full credit.

Late Work Deadline: Nov 12, 2012

Chapter 3

GradeBot

Contents

`http://gbot.dc.is2.byuh.edu/` is the web interface for what used to be a huge system called GradeBot. What you see at that URL is called GradeBot Lite.

GradeBot is an automated program grader.

3.1 Testing Your Program

You write your program and upload it or paste it or key it into GradeBot. Then you press a button to have it graded. GradeBot will tell you if your program works properly or not.

If your program works properly, you will see this message:

```
Success!  No errors found.  Nice job.  Assignment complete!
```

If your program does not work properly, you will see this message:

```
Please fix your program and submit it again.
```

3.2 How GradeBot Tests

The way GradeBot works is that it has a version of each program you are asked to write. It generates some sample inputs, runs its own version, and collects the outputs. That is used to make a script. Your program is expected to behave exactly according to the script.

This makes some serious demands on your program. You must get all the strings right. If GradeBot wants `"Please enter a number: "` then that's exactly what your program must print. You may find yourself squinting at the output where GradeBot says you missed something.

Once you get the first test right, GradeBot typically invents another test and has you run it. And another. And another. Eventually, you either make a mistake, or you get them all correct.

If you make a mistake, GradeBot will tell you what it was expecting, and what it got instead.

If you get everything correct, GradeBot will announce your success.

3.3 Submitting Your Program

Once GradeBot says your program behaves correctly, you can submit it to me, the instructor.

GradeBot does not inform me about the success or failure of your program, nor how many attempts you made. It only reports to you.

Currently I require you to send your program as an email message.

The subject line must begin with the task name as given by GradeBot.

The body of the message must be your program, as though you had done a cut-and-paste job to send it to me.

I do not accept programs sent as attachments because of the extra work it requires on my end.

GradeBot does not care about style or comments. I tend to care about both. So, when GradeBot says your program is okay, it means it runs okay. It may still need some improvements.

Interpreting GradeBot's Requirements

Quotes are shown in the examples to delimit the contents of the input and output lines. The quotes themselves are not present in the input, nor should they be placed in the output.

Each line ends with a newline character unless specified otherwise.

Each line ends with a newline character unless specified otherwise.

I said that twice because it is one of the most common mistakes students make.

Numbered lines are shown to designate output that your program must create.

```
"Hello"
```

If GradeBot expects the line `"Hello"` then you must print the line `"Hello\n"`. You must add the `\n` to indicate that the line is complete.

```
"Hello" (no \n)
```

If GradeBot expects the line `"Hello" (no \n)` then you must print the line `"Hello"` without any `\n`.

```
""
```

This means that GradeBot expects a blank line. You must print `"\n"` to make that happen.

```
in> .......
```

`"in>"` is shown to designate input that your program will be given (through the standard input channel).

```
eof  (end of output)
```

`eof` means end of file, and indicates that your program must terminate cleanly.

Chapter 4

q1: String Basic

These points are earned during a final exam (or early final) by writing a working and correct program that does what is required.

The key things to demonstrate here are:

(a) How to get string input into your program. This is done by reading from `<STDIN>` and storing the result in a variable.

Example: `$flavor = <STDIN>;`

(b) How to remove the newline from the end of the string. This is done by using the `chomp` command.

Example: `chomp ( $flavor );`

(a) and (b) are often combined into a single statement.

Example: `chomp ( $flavor = <STDIN> );`

(c) How to compose a printed statement that includes information from your variables. This is done by using the variable name within another string.

Example: `print "I love $flavor ice cream."`

(d) Do exactly what was requested. If I request specific wording, you must follow it exactly. If I do not specify something exactly, you are free to do anything that works.

Example: `print "I love $flavor ice cream. "`

In this example, there is a space after ice cream. If my specification says there should be no space, then by putting a space you will lose credit for your work.

Chapter 5

q2: Number Basic

These points are earned during a final exam (or early final) by writing a working and correct program that does what is required.

You should already have the skills involved with String Basic.

The key thing to demonstrate here is:

(a) How to use simple arithmetic to calculate an answer.

Example: `$x = 2 * $y - 5;`

You will be told specifically what to do. For example, read in two numbers, multiply them together, and then add 5.

Parentheses may be useful in getting formulas to do the right thing.

Note: it is usually not necessary to `chomp` inputs that are numbers. Perl will still understand the number fine.

Chapter 6

q3: Number Story

These points are earned during a final exam (or early final) by writing a working and correct program that does what is required.

You should already have the skills involved with Number Basic.

Story problems are problems where the precise steps are not given to you. Instead, you must understand the problem and develop your own formula. Sometimes this is easy. Sometimes this is difficult.

The main thing we are measuring is whether you can invent your own formula based on the description of the problem.

Remember to test your program. Make sure your formula gives correct answers.

Chapter 7

Style Requirements

As your programs become more complex, style becomes important.

In real life programming situations, it is common for work groups to adopt style rules. By using the same style, programs tend to be easier to read and understand. For most of the problems on each test, specific style is required.

Because style is a huge aid to making your program easier to read, I have developed the following style rules.

Spacing

The first style rule I require is spacing. I am very picky. You must put one space between tokens. There are a few exceptions.

Example: `(3+5)` is bad.

Example: `( 3 + 5 )` is good.

This requires that you know what a token is. I cover this in the text book.

Mistake: adding spaces inside a quoted string changes its meaning. A quoted string is by itself a single token. I require spaces between tokens, not within tokens.

Exception: You may omit the space before a semi-colon.

Example: `$x = ( 3 + 5 );` is okay.

Exception: You may omit the space between a variable and a unary operator.

Example: `$x++;` is okay.

Example: `$x = -$y;` is okay.

Use the Values Specified

Often a problem will specify certain numbers or strings that define how the program should run. If possible, use those exact same values in writing your program. If not, include a comment that has the exact value.

Example: Print "Hello, World!"

Good: `print "Hello, World!"`

Okay: `print "Hello, World!\n"`

Bad: `print "hello, world!"` (wrong capitalization)

Bad: `print " Hello, World! "` (extra spaces)

Example: Print the numbers from 1 to 100.

Good: `for ( $i = 1; $i <= 100; $i++ ) { print $i }`

Bad: `for ( $i = 1; $i < 101; $i++ ) { print $i }`

If you cannot use the exact value specified in your program itself, then use the exact value in a comment nearby.

Example: If the last name is in the A-G range, do something.

Good: `if ( uc $ln lt "H" ) { # A-G`

Mathematical Parentheses

In the form `$x = ( something );` there is a confusing ambiguity. I do not allow it because it is confusingly ambiguous.

The problem is ambiguity. It has two possible meanings. Perl probably handles it okay, but I still do not accept it.

Parentheses can be used in a **mathematical expression** to force a certain order of operations.

Example: `$x = ( 3 + 2 ) * 5; # this is okay`

Example: `$x = ( ( 3 + 2 ) * 5 ); # this is not okay`

Parentheses are also used in **defining arrays.**

Example: `@x = ( 3 ); # this is okay`

Here is the ambiguity that we wish to avoid:

Example: `$x = ( 3 ); # $x will be 3`

Example: `$x = @x = ( 3 ); # $x will be 1`

Bottom line? Do not put parentheses around a whole expression or statement. If you do, I will probably mark it wrong.

One Statement Per Line

Each statement should be on its own line.

In real life, statements are often combined onto one line if they are closely related. This is not real life. For exams, it is easier if I have a simple rule and stick with it.

Start a new line after each opening { or semi-colon.

Exception: The `for` loop uses two semi-colons to separate its control structure (init; condition; step). You should not normally start a new line after those semi-colons.

Exception: A relevant comment can be placed after a semi-colon.

Indenting

Indent is the number of blanks at the start of each line.

The main program should not be indented. There should be no spaces in front of the actual code.

Blocks are created by putting { before and } after some lines of code. This happens with decisions, loops, and subroutines.

Within the block, I require indenting to be increased by two.

Warning: because crazy indenting makes programs substantially harder to read, I have become very picky about this.

Warning: If you write your program using an editor like notepad++, and then cut-and-paste it to save as your exam answer, the indenting may be messed up. You should go back through your program and fix any indenting problems that may have occurred.

Common Error: using TAB instead of two spaces. I will mark it wrong.

Common Error: using one space instead of two spaces. I will mark it wrong.

Helpful Blank Lines

Blank lines are used to divide a program into natural "paragraphs." The lines within each paragraph are closely related to each other, at least as seen by the programmer.

Rule: Keep things fairly compact. Use blank lines and comments to help visually identify groups of related lines. Do not use an excessive number of blank lines.

Helpful Names

Variables and subroutines are named. The computer does not care how meaningful the names are that you use, but programmers will care. I will care. The names should be helpful. They should bear some obvious relationship to the thing they represent.

Long descriptive names can be abbreviated and explained when used.

Example: `$eoy = 1; # eoy means end of year, 1 means true.`

Names like $x and $y should be avoided because they normally don't convey meaning. Like "he", "she", and "it" in English, their meaning is short-range and would need to be clear by the immediately surrounding context.

Chapter 8

q4: Number Decision

These points are earned during a final exam (or early final) by writing a working and correct program that does what is required.

The emphasis here is on decision. How do you decide what to do? How do you express your desires?

The key things to demonstrate here are:

(a) How to write an `if` statement.

(b) How to compare two numbers. This includes:

(b1) Example: `( $x < $y )` means less than.

(b2) Example: `( $x <= $y )` means less than or equal to.

(b3) Example: `( $x == $y )` means equal to.

(b4) Example: `( $x > $y )` means greater than.

(b5) Example: `( $x >= $y )` means greater than or equal to.

(b6) Example: `( $x != $y )` means not equal to.

(c) Near Misses. Things that look right but are wrong.

(c1) Example: `( $x = $y )` is a frequent typo for equal to, but actually means "gets a copy of".

(c2) Example: `( $x => $y )` is a frequent typo for greater than or equal to, but means the same thing as comma does when defining an array.

Chapter 9

q5: Number Decision Story

These points are earned during a final exam (or early final) by writing a working and correct program that does what is required.

As with number story, we have a story problem. And a decision will be involved. You will need to analyze the question and decide how to solve it.

Chapter 10

q6: String Decision

These points are earned during a final exam (or early final) by writing a working and correct program that does what is required.

The emphasis here is on strings and how their decisions differ from numbers.

The key things to demonstrate here are:

(a) How to compare two strings. This includes:

(a1) Example: `( $x lt $y )` means less than.

(a2) Example: `( $x le $y )` means less than or equal to.

(a3) Example: `( $x eq $y )` means equal to.

(a4) Example: `( $x gt $y )` means greater than.

(a5) Example: `( $x ge $y )` means greater than or equal to.

(a6) Example: `( $x ne $y )` means not equal to.

(b) Near Misses. Things that look right but are wrong.

(b1) Example: `( $x eg $y )` is a frequent typo for eq.

(c) Properly quote your literal strings. (See barewords in the text book.)

(c1) `( $x eq "hello" )` is the right way to quote a string.

(c2) `( $x eq hello )` is the wrong way to quote a string.

(c3) `( $x eq $y )` is right because it is a variable, not a literal.

Chapter 11

q7: String Bracket

These points are earned during a final exam (or early final) by writing a working and correct program that does what is required.

The emphasis here is on more complicated decisions, where there are more than two options.

The key things to demonstrate here are:

(a) How to handle "clarinet through costly".

(b) How to handle "a-j, k-o, p-z".

(c) How to handle all possible capitalizations.

(d) Properly quote your literal strings. (See barewords in the text book.)

(d1) `( $x eq "hello" )` is the right way to quote a string.

(d2) `( $x eq hello )` is the wrong way to quote a string.

(d3) `( $x eq $y )` is right because it is a variable, not a literal.

Chapter 12

q8: Repeat While

These points are earned during a final exam (or early final) by writing a working and correct program that does what is required.

Loops are an important tool. Repeat While names a specific instance of that.

The syntax is `while ( condition ) { block }`

In these loops the condition is just like `if` statements have. Often it is a comparison like `( $x < 100 )` .

The block is the collection of commands that will be done repeatedly, so long as the condition is still true.

Common error: make sure the condition will eventually become false. If your condition checks for `$x` less than 100, make sure that `$x` is changing and will eventually reach 100.

Common error: if the ending condition gets skipped, the loop could run forever. `( $x < 100 )` is much safer than `( $x != 100 )` .

Common error: confusing the `while` syntax with the `for` syntax.

Chapter 13

q9: Repeat For

These points are earned during a final exam (or early final) by writing a working and correct program that does what is required.

Loops are an important tool. Repeat For names a specific instance of that.

The syntax is `for ( init; condition; step ) { block }`

The `init` part initializes the variable that controls the loop.

The `condition` part is just like a `if` statement or `while` statement.

The `step` part is usually something like `$x++` that increments the control variable.

Common error: confusing the `while` syntax with the `for` syntax.

Chapter 14

q10: Repeat Last

These points are earned during a final exam (or early final) by writing a working and correct program that does what is required.

Loops are an important tool. Repeat Last names a specific instance of that.

The syntax is `while ( 1 ) { block }` where the block includes something like this to break out of the loop:

```
if ( condition ) { last }
```

Common error: due to style requirements, `last` should be on a new line, properly indented.

Chapter 15

q11: Repeat Nested

These points are earned during a final exam (or early final) by writing a working and correct program that does what is required.

Loops are an important tool. Repeat Nested names a specific instance of that.

What we are looking for here is the ability to run one loop (the inner loop) inside another loop (the outer loop).

Example: print all possible combinations for a child's bike lock, where there are four wheels each ranging from 1 to 6.

Chapter 16

q12: List Basic

These points are earned during a final exam (or early final) by writing a working and correct program that does what is required.

Lists and arrays are the same thing. When we talk about lists, we are not using indexing. When we talk about arrays, we are using indexing.

The key things to demonstrate here are:

(a) An array can be initialized by listing elements in parentheses.

Example: `@x = ( "cat", "dog", "bird" );`

(b) An array can be modified.

(b1) using `push` to add something to the back end of a list.

Example: `push @x, "hello";`

(b2) using `pop` to remove something from the back end of a list.

Example: `$x = pop @x;`

(b3) using `shift` to remove something from the front end of a list.

Example: `$x = shift @x;`

(b4) using `unshift` to add something to the front end of a list.

Example: `unshift @x, "hello";`

Chapter 17

q13: List Loop

These points are earned during a final exam (or early final) by writing a working and correct program that does what is required.

The key thing to demonstrate here is how to use a foreach loop.

Example: `foreach $book ( @books ) { print $book }`

Example: `foreach ( @books ) { print $_ }`

Wrong: `foreach @books { print $_ }`

Chapter 18

q14: Array Basic

These points are earned during a final exam (or early final) by writing a working and correct program that does what is required.

Lists and arrays are the same thing. When we talk about lists, we are not using indexing. When we talk about arrays, we are using indexing.

(a) The whole array is named with `@` at the front.

(b) Individual slots in the array are named with `$` at the front, and `[number]` at the back.

(c) The first item in an array is at location zero.

Example: `$x = $array[0];`

Example: `$array[0] = $x;`

(d) The second item in an array is at location one.

Example: `$x = $array[1];`

(e) The last item in an array is at location -1.

Example: `$x = $array[-1];`

(f) The second to last item in an array is at location -2.

Example: `$x = $array[-2];`

Ambiguous: `@x[1]` - Perl accepts it for `$x[1]` but I do not.

Chapter 19

q15: Array Loop

These points are earned during a final exam (or early final) by writing a working and correct program that does what is required.

The key thing to demonstrate here is how to use a for loop.

(a) The size of an array can be found out.

Example: `$size = @array;`

(b) A `for` loop can be used to "index" your way through an array.

Okay: `for ( $i = 0; $i < $size; $i++ ) { print $array[$i] }`

Wrong: `for ( $i = 0; $i <= $size; $i++ ) { print $array[$i] }`

Okay: `for ( $i = 0; $i < @array; $i++ ) { print $array[$i] }`

Chapter 20

q16: Array Split

These points are earned during a final exam (or early final) by writing a working and correct program that does what is required.

The `split` command can be used to convert a string into an array.

Example: `@x = split ":", "11:53:28";`

Common mistake: `$x = split ...` (because dollar-x should be at-x)

Chapter 21

q17: Array Join

These points are earned during a final exam (or early final) by writing a working and correct program that does what is required.

The `join` command can be used to convert an array into a string.

Example: `$x = join ":", ( "11", "53", "28" );`

Common mistake: `@x = join ...` (because at-x should be dollar-x)

Chapter 22

Subroutine Basics

All subroutine points require you to do the basic elements of each subroutine correctly.

Subroutines are defined using the following syntax:

```
sub name { block }
```

The word `sub` must be given first. It is not `Sub` or `subroutine` or forgotten.

Never use global variables unless they are necessary. That means each variable in a subroutine should be introduced with the word `my` the first time it appears, unless you are sure it is supposed to be global.

Exception: `@_` in a subroutine is naturally local. You don't have to `my` it.

Chapter 23

q18: Subroutine Return

These points are earned during a final exam (or early final) by writing a working and correct program that does what is required.

(a) To return a single number from a subroutine, you can do it like this.

Example: `return 5;`

Example: `return $x;`

Wrong: `return ( 5 );` - this is an ambiguity error.

(b) To return a string from a subroutine, you can do it like this.

Example: `return "this is a string";`

Wrong: `return ( "this is a string" );` - ambiguity.

(c) To return an array from a subroutine, you can do it like this.

Example: `return ( 1, 2, 4, 8 );`

Wrong: `return "( 1, 2, 4, 8 )";` - a string is not an array

Example: `return ( "this", "is", "a", "list" );`

Wrong: `return ( this, is, a, list );` - each string should be quoted

Example: `return @x;`

Wrong: `return "@x";` - a string is not an array

(d) `return` and `print` do different things. Return gives something back to the caller. Print sends something to the end user.

Chapter 24

q19: Positional Parameter

These points are earned during a final exam (or early final) by writing a working and correct program that does what is required.

We are testing your ability to retrieve parameters that were passed into a subroutine.

The arguments to a subroutine arrive in the local variable `@_` and can be retrieved from it.

Positional parameters are always in the same slot of the array. You can get the third positional parameter by using `$_[2]` for example.

Chapter 25

q20: Globals and Locals

These points are earned during a final exam (or early final) by writing a working and correct program that does what is required.

We are testing your ability to maintain privacy on the variables you use in your subroutine.

In Perl, variables are naturally global. This is now widely recognized to be a bad thing, but it is too late to change now.

To force variables to be local (which is the opposite of global), you have to specially mention the word `my` before the variable the first time it is used.

Example: `my $abc;` - creates a local variable named `$abc`.

Example: `my ( $abc );` - creates a local variable named `$abc`.

Example: `my ( $abc, $def, $ghi );` - creates three local variables named `$abc`, `$def`, `$ghi`, respectively.

Common Error: `my $abc, $def, $ghi;` - creates ONE local variable named `$abc`, and mentions two global variables named `$def` and `$ghi`.

Chapter 26

q21: Variable Number of Parameters

These points are earned during a final exam (or early final) by writing a working and correct program that does what is required.

We are testing your ability to retrieve parameters that were passed into a subroutine.

The arguments to a subroutine arrive in the local variable `@_` and can be retrieved from it.

A `foreach` loop is usually used to walk through the list of parameters that were sent to the subroutine.

Chapter 27

Final Projects

These points are earned outside of class by doing a final project.

Doing a project is a great way to become empowered. Our nominal goal is that each student be able to build something fun and useful. The real goal is to enrich the student by giving them the ability to create the programs they need without always relying on others.

Final projects must be different from things we did in class. They can be similar, but should have at least a few fundamental improvements or changes. Simply using a different picture or different words is not enough. It must have different logic.

Size

What is the right size for a project at this point in your skills development? This unit contains a few pre-defined projects that could be appropriate for demonstrating and improving your programming skills. They are given as examples. They have served in the past as actual assignments.

Trust

Because out-of-classroom projects by their nature are done without supervision, there is some risk that students will get inappropriate help. To guard against this, project points can only be earned by students who have already performed sufficiently well on the in-class exams and activities.

Normally this means you must have already earned a B- through your other work.

You can do the project even before you have earned a B- but it will not be counted until you earn a B-.

Invent a Project

Doing pre-defined projects can be boring and can lead to some inappropriate sharing of code. This does not enhance learning. So instead here is a list of requirements that your project should satisfy.

Online: It must run online as a web application. Anyone in the world should be able to run your program.

Authorship: The code comments and the program output should clearly identify you as the author and owner of the program.

Creative: Do something creative and unique. If it looks like the project your neighbor already turned in, it might not qualify. If it is too similar to something we did in class, it would not qualify.

Fun: Your program should be fun. A game would be ideal. Fun is a subjective judgment, so we will trust you on this. If you think it is fun, we will agree that it is fun.

Images: The program should appropriately use pictures, typically by way of an HTML `<img>` statement. Ideally the pictures would change depending, for example, on the progress of the game.

Input: Your program must accept multiple inputs, for example buttons or text fields, to allow the user to interact with it. Hidden fields count as inputs.

State: Your program must have some sort of meaningful state that it carries forward. Some or all of the state must be carried in hidden fields that are actually important to your program's operation.