CIS 101 Study Guide
Winter 2013

Don Colton
Brigham Young University–Hawaii

April 4, 2013

This is a study guide for the CIS 101 class, Introduction to Programming, taught by Don Colton, Winter 2013.

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

Overview

Contents

For many it is a really fun class. Others hate it. This class can be hard work.

It is a foundational step in developing your ability to serve those around you by giving them better ways to use their computers.

And it can be the gateway to automating some of the (mind-numbing) tasks that can be involved with things like computer systems administration.

We build web-based programs that you can share through the Internet with anyone in the world: friends, family, anybody. And we develop skills you can use in later classes and the work place.

The textbook is online free. You can download the PDF from `http://ipup.doncolton.com/`

We will use the Third Edition. I often make small improvements, like spelling corrections, throughout the semester. Once in a while I make bigger changes, like whole new sections or chapters. So printing it out is not really recommended. Why kill a tree? Reading a fresh copy on your computer would be ideal, if that works for you.

I try hard to not lecture much in class. (Sometimes I end up lecturing anyway.) The book contains my lectures. You read them outside of class. In class we may review parts of the reading and I always give you a chance to ask questions.

Most time in class is spent actually making things. I go over parts of the textbook to introduce activities, but there is lots more in the book that you are expected to read on your own.

Based on past experience, everyone who regularly attends will pass the class. (The F grades usually only happen to people who quit coming to class.)

Most students do a project. The study guide has official project details.

1.1 So, What is Programming?

First the bad news. Computers are pretty stupid. You have to give them simple steps to follow.

Now the good news. Computers are fast, reliable, and cheap. They don't get offended, go on strike, call in sick, or take vacation.

Many interesting tasks can be built up out of the simple steps that computers can perform. For these reasons, even though they are pretty stupid, computers are very popular.

The art of programming is the art of converting useful activities into simple steps that a computer can perform.

Our programming language will be Perl.

1.2 Course Learning Objectives

This study guide and the accompanying textbook are intended to meet the following learning objectives.

By the conclusion of this course, students will demonstrate the ability to write clear and correct programs that utilize the following techniques.

- sequences of simple steps
- simple variables
- decisions (if, else, elsif)

- looping (while, for, foreach)
- array and list variables
- subroutines

Those are the major skills. In teaching those, I also teach the following:

- dynamic web page creation
- dynamic response to web page inputs

Students will demonstrate most of these skills by creating short programs that perform specific tasks in timed and supervised situations.

1.3 Reporting Your Study Time

Once a week I will ask for your study time as part of the daily update.

For study during the very last week, which includes the final exam, you can report on the day of the final exam.

If you do not report in some other way, you can report by sending me the details by email.

With email, use this as your subject line:

`cis101 study time (lastname firstname)`

Inside the email, say something like: "For the week of (starting month and day) to (ending month and day), I studied (how many) hours." Be specific about which week it is.

Chapter 2

Activities In General

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.

Each activity will be discussed in class, and will have a due date, a deadline, and a grading label.

Discussed means the date we talked about it in class.

Due date means the date by which you must complete the assignment for it to be "on time." After that it is late, but it is still accepted for full credit until the deadline.

23:59 means 11:59 PM.

Deadline means the date by which you must complete the assignment to receive full credit for your work. After that it is not accepted for credit.

If you are submitting something after the due date but before the deadline, you must either submit it by email or else notify me by email so I know to grade it.

Often the deadline is two weeks after the due date.

Grading Label means a short label I use to track this activity for grading purposes. Online activities have labels that start with o. GradeBot activities have labels that start with g. Command-line activities have labels that start with c.

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

Grading Rules (Rubric) means the specific rules by which grades are decided for each activity.

2.1 Email Submission Rules

In some cases, I require you to submit your work by email. When email is involved, there are a few rules I need you to follow.

If your program violates the rules enough that grading becomes difficult, I will probably reply to your submission telling you what rules you violated and asking you to fix and resubmit.

2.1.1 To: Line

You can email to `doncolton2@gmail.com`.

You can email to `don.colton@byuh.edu`.

They both ultimately go the same place, so you do not need to send to both. Either one is fine.

2.1.2 Subject Line

The subject line of the email must be as follows:

```
cis101 gradinglabel lastname firstname
```

The reason for this rule is to facilitate the recording of grades. When I receive your email, it may be in the midst of many other emails from other students. I need to keep things straight so that I can record your grade properly.

The `gradinglabel` part is replaced by the grading label for that assignment.

The `lastname` part is replaced by your own last name.

The `firstname` part is replaced by your own first name. This is the name that you asked me to use for you. I use that name in my grade book.

When I go to record your grade, I scan down my grade book, which is sorted by lastname and firstname. If I do not see the lastname and firstname that you provided, it requires extra steps for me to verify which person should receive credit. I would prefer to have you do those extra steps instead of me.

So, for example, if I were submitting task p1 and my lastname were Colton and my firstname were Don, I would use this subject line:

```
cis101 p1 Colton Don
```

I am not picky about things like dots and commas, so I would also accept this subject line:

```
cis101 p1 Colton, Don
```

2.1.3 Body When Submitting A Program

If you are submitting a program, the remainder of the email should be that program, and nothing else unless the assignment specifically requires it.

Do not send your program as an attachment. Send it directly in-line as plain text so I will see it immediately when I open your email.

The first line of your program must be a comment line that repeats the required subject line. This is to facilitate the recording of grades.

In Perl, comment lines start with `#`. So, following the previous example from above, I would have this comment line as the first line of my program.

```
# cis101 p1 Colton Don
```

Optionally, you can include this line, or one that is substantially the same, before your "subject line" comment:

```
#! /usr/bin/perl
```

The email must be in plain-text form. It should not be in html form or rich text form or in the form of an attachment. In plain text, there is no coloring to the letters. There is no bold or italics.

The reason for this rule is to facilitate testing of your program. When I receive your email, I may need to test it by running it. I do this by doing a copy-paste from your email into GradeBot (for instance) or into an empty program file. Then I run your program.

I should be able to use copy-paste to make a copy of your program for testing.

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

You must avoid having each line of your program start with `>` or `>>` as is common when you are replying. Having those characters makes it impossible to copy-paste and run your program.

If your program includes any long comments, make sure each line of the long comment starts with the `#` character. Otherwise, the program will not run. I mention this because your email client may automatically break up long lines, thus converting your correct and working program into an incorrect and broken program. Be alert to this possibility and protect against it by not using long lines.

You should avoid having anything else in your email that might make it difficult for me to decide what you intend as your program. I want to be able to assume that your whole email is the program your are submitting.

Specifically avoid including any "reply" comments, but if they are obviously not part of the program, they will be okay.

Specifically avoid a "signature", but if it are obviously not part of the program, it will be okay.

2.2 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 (index.html) or CGI program (index.cgi) that is properly linked to the CIS 101 student projects page. It should clearly display 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. Follow the email rules in section 2.1 (page 7) in the construction of your subject line.

Rubric: Typical Grading Rules for a one-day task:

Grade 0: Missing, or fails to run.

Grade 1: Partly works but fails one or more requirements.

Grade 2: (normal) Meets all requirements.

Grade 3: (rare) Is particularly impressive.

2.3 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.

Follow the email rules in section 2.1 (page 7) as you construct your subject line and the body of your message.

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.

2.4 gXX: GradeBot Task General Rules

GradeBot itself is explained in chapter 5 (page 50).

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.

Currently the labels are numeric digits.

Lab ID: Within GradeBot, each task has a lab ID (name). Normally this ID consists of "cis101.(label).(word)" where (label) is the label above (with or without the g prefix), and (word) helps identify and describe 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.

Follow the email rules in section 2.1 (page 7) as you construct your subject line and the body of your message.

Specifically, your subject line should look like this:

`cis101 gxx lastname firstname` is the required subject line.

where gxx is replaced by the grading ID number, lastname is replaced by your lastname as shown on my roll sheet, and firstname is replaced by your firstname as shown on my roll sheet. (You can put a comma between lastname and firstname if you want.)

The body of your email should be your program and nothing else, in plain text. The first line of your program must be a comment line (in Perl) that repeats the subject line but with a hash in front, like this:

`# cis101 gxx lastname firstname` is the required comment line.

Is it difficult for me to tell what part of your email is your program? Submit only your program, and not anything else.

Is your program not copy-paste ready? For example, does it have `>` at the front of any of the lines? It should not.

Is your program in plain text or rich (html) text? It should be in plain text.

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 I tell you to 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. You may want to save my reply until you see your grade reflected in my gradebook.

Normal 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.

Chapter 3

Activities Assigned

Contents

This is a list of the activities that have been assigned. They are listed in the order they were discussed in class.

As new tasks are assigned, they will be added to this chapter.

The purpose of these activities is to give you a way to develop and test your programming skills.

3.1 g12: Hi Fred

- Status: Officially Assigned.
- Discussed: Wed, Jan 9.
- Due Date: Thu, Jan 10, 23:59
- Deadline: Tue, Jan 15, 23:59
- Grading Label: **g12**

This is a GradeBot task. The general rules in section 2.4 (page 11) apply, including email subject line and program comment line.

`cis101 g12 lastname firstname` is the required subject line.

`# cis101 g12 lastname firstname` is the required comment line.

Gradebot can be found at `http://gradebot.tk/` and at `http://gbot.dc.is2.byuh.edu/`.

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

Sample Execution:

```
GradeBot would have engaged your program in this dialog:
note  GBot "# debug output lines are permitted"
  1:  "What's your name?"
 in>  "Fred"
  2:  "Hello, Fred!"
 eof  (end of output)
```

3.2 cML: Mad Lib

- Status: Officially Assigned.
- Discussed: Fri, Jan 11.
- Due Date: Mon, Jan 14, 01:00
- Deadline: Tue, Jan 15, 23:59
- Grading Label: **cML**

This is a command-line task. The general rules in section 2.3 (page 10) apply, including email subject line and program comment line.

`cis101 cML lastname firstname` is the required subject line.

`# cis101 cML lastname firstname` is the required comment line.

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. Test your program. Then email it to me.

Requested: Please make the story creative and interesting.

3.3 oP1: First Web Page

- Status: Officially Assigned.
- Discussed: Mon, Jan 14.
- Due Date: Tue, Jan 15, 23:59
- Deadline: Thu, Jan 17, 23:59
- Grading Label: **oP1**

This is an online task, but you do not need to send me the code that you write. It just needs to work when I test it online. The general rules in section 2.2 (page 9) apply.

If you are requesting a regrade, this is the required subject line:

```
cis101 regrade oP1 lastname firstname
```

Task: Create a static web page (index.html) that is (a) properly linked from the student projects page. It must include (b) your name, and (c) CIS 101.

3.4 oDi: Dice

- Status: Officially Assigned.
- Discussed: Wed, Jan 16.
- Due Date: Thu, Jan 17, 23:59
- Deadline: Thu, Mar 7, 23:59
- Grading Label: **oDi**

This is an online task, but you do not need to send me the code that you write. It just needs to work when I test it online. The general rules in section 2.2 (page 9) apply.

If you are requesting a regrade, this is the required subject line:

```
cis101 regrade oDi lastname firstname
```

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.

For each roll, give a numeric or written explanation, and give an appropriate image.

Example: Roll two dice. Assume they come up 2 and 5. Show the number 2 and a picture of the "2" face of a die. Show the number 5 and a picture of the "5" face of a die.

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

3.5 oML: Mad Lib Online

- Status: Officially Assigned.
- Discussed: Wed, Jan 23.
- Due Date: Thu, Jan 24, 23:59
- Deadline: Mon, Jan 28, 00:01
- Grading Label: **oML**

This is an online task, but you do not need to send me the code that you write. It just needs to work when I test it online. The general rules in section 2.2 (page 9) apply.

If you are requesting a regrade, this is the required subject line:
`cis101 regrade oML lastname firstname`

Task: Create a MadLib web page.

As always, your program must be properly linked to the student projects page, and the displayed webpage must show your name.

(a) Your program must display and accept three or more input fields. Each must have a suitable description and a reasonable (non-blank) default value.

The first time your program runs, it must use your default values to construct the story.

(b) Your program must have a submit button. When the submit button is pressed, the screen should be redrawn. The input values should still be as entered. A story must be presented that uses the contents of the input fields.

(c) We provide the following subroutine to make this easier.

You can place a copy of this subroutine at or near the beginning or end of every program you write that requires online inputs. You are welcome to use copy-paste to insert it into your program. Look under "olin" in the index of the textbook for an explanation of this subroutine.

The olin Subroutine:

```
sub olin { my ( $name, $res ) = @_;
  if ( $_olin eq "" ) { $_olin = "&" . <STDIN> }
  if ( @_ == 0 ) { return $_olin }
  if ( $_olin =~ /&$name=([^&]*)/ ) {
    $res = $1; $res =~ s/[+]/ /g;
    $res =~ s/%(..)/pack('c',hex($1))/ge }
  return $res }
```

Using olin:

Include the olin subroutine in your program. It can be anywhere in your program, top or bottom or in between. Then include one or more calls to olin as shown here.

You normally call olin with two parameters, like this:

`$x = olin ( "name", "default" );`

Notice that `chomp` is not needed and is not used with olin.

In this case, olin searches the inputs that were sent by the form on your webpage, and returns the value of the first field whose name is "name." If there is no such field, olin returns the value you provides as "default."

You can call olin with one parameter, like this:

`$x = olin ( "name" );`

In this case, if there is no matching field, olin returns the "undefined" value.

You can call olin with no parameters, like this:

`$x = olin ( );`

In this case, olin returns the entire input string that was sent by the browser, with `&` added to the front.

3.6 g16: Celsius

- Status: Officially Assigned.
- Discussed: Fri, Jan 25.
- Due Date: Fri, Jan 25, 23:59
- Deadline: Tue, Jan 29, 23:59
- Grading Label: **g16**

This is a GradeBot task. The general rules in section 2.4 (page 11) apply, including email subject line and program comment line.

`cis101 g16 lastname firstname` is the required subject line.

`# cis101 g16 lastname firstname` is the required comment line.

Summary: Convert temperature in Fahrenheit to Celsius.

The formula is: celsius = (fahrenheit - 32) * 5 / 9

Fix and resubmit?

After you submit, if I say "Incomplete. Fix and resubmit." check these things.

Is it difficult for me to tell what part of your email is your program? Submit only your program, and not anything else.

Is your program not copy-paste ready? For example, does it have > at the front of any of the lines? It should not.

Is your program in plain text or rich (html) text? It should be in plain text.

Sample Execution:

```
GradeBot would have engaged your program in this dialog:
note  GBot "# debug output lines are permitted"
  1:  "Enter a temperature in Fahrenheit: " (no \n)
 in>   .................................."77"
  2:  "77 in Fahrenheit equals 25 in Celsius."
 eof  (end of output)
```

3.7 g21: Numeric Decision

- Status: Officially Assigned.
- Discussed: Mon, Jan 28.
- Due Date: Tue, Jan 29, 23:59
- Deadline: Thu, Mar 7, 23:59
- Grading Label: **g21**

This is a GradeBot task. The general rules in section 2.4 (page 11) apply, including email subject line and program comment line.

`cis101 g21 lastname firstname` is the required subject line.

`# cis101 g21 lastname firstname` is the required comment line.

Summary: Compare two numbers. Print an appropriate response.

Sample Execution:

```
GradeBot would have engaged your program in this dialog:
note  GBot "# debug output lines are permitted"
  1:  "How much money do you have? " (no \n)
 in>   ............................"1.00"
  2:  "How much does the gift cost? " (no \n)
 in>   .............................."2.00"
  3:  "Sorry. You cannot afford it."
 eof  (end of output)
```

3.8 g22: String Decision

- Status: Officially Assigned.
- Discussed: Wed, Jan 30.
- Due Date: Thu, Jan 31, 23:59
- Deadline: Thu, Mar 7, 23:59
- Grading Label: **g22**

This is a GradeBot task. The general rules in section 2.4 (page 11) apply, including email subject line and program comment line.

`cis101 g22 lastname firstname` is the required subject line.

`# cis101 g22 lastname firstname` is the required comment line.

Summary: Compare a word to two other words. Tell whether it is before, between, or after those words.

Sample Execution:

```
GradeBot would have engaged your program in this dialog:
note  GBot "# debug output lines are permitted"
  1:  "Page 20 of the phone book starts with Davis and ends with Dodson."
  2:  "What name do you seek? " (no \n)
 in>   ......................"Ditto"
  3:  "It would be on page 20."
 eof  (end of output)
```

3.9 oF1: Farm 1

- Status: Officially Assigned.
- Discussed: Fri, Feb 8.
- Due Date: Mon, Feb 11, 09:00
- Deadline: Thu, Mar 7, 23:59
- Grading Label: **oF1**

This is an online task, but you do not need to send me the code that you write. It just needs to work when I test it online. The general rules in section 2.2 (page 9) apply.

If you are requesting a regrade, this is the required subject line:
`cis101 regrade oF1 lastname firstname`

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.

3.10 g31: While Loop

- Status: Officially Assigned.
- Discussed: Mon, Feb 11
- Due Date: Tue, Mar 5, 23:59
- Deadline: Tue, Mar 12, 23:59
- Grading Label: **g31**

This is a GradeBot task. The general rules in section 2.4 (page 11) apply, including email subject line and program comment line.

`cis101 g31 lastname firstname` is the required subject line.

`# cis101 g31 lastname firstname` is the required comment line.

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).

This matches one of the problems that is on the final exam.

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. I will copy-paste your work into GradeBot. The proper spacing and indenting must be evident after I paste it in.

Sample Execution:

```
GradeBot would have engaged your program in this dialog:
note  GBot "# debug output lines are permitted"
  1:  "Where should I start? " (no \n)
 in>   ......................"1"
  2:  "Where should I end? " (no \n)
 in>   ...................."5"
  3:  "What should I count by? " (no \n)
 in>   ......................."1"
  4:  "1"
  5:  "2"
  6:  "3"
  7:  "4"
  8:  "5"
  9:  "Done!"
 eof  (end of output)
```

3.11 g33: Last Loop

- Status: Officially Assigned.
- Discussed: Wed, Feb 13
- Due Date: Tue, Mar 5, 23:59
- Deadline: Tue, Mar 12, 23:59
- Grading Label: **g33**

This is a GradeBot task. The general rules in section 2.4 (page 11) apply, including email subject line and program comment line.

`cis101 g33 lastname firstname` is the required subject line.

`# cis101 g33 lastname firstname` is the required comment line.

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.

Sample Execution:

```
GradeBot would have engaged your program in this dialog:
note  GBot "# debug output lines are permitted"
  1:  "Where should I start? " (no \n)
 in>   ......................"1"
  2:  "Where should I end? " (no \n)
 in>   ...................."5"
  3:  "What should I count by? " (no \n)
 in>   ........................"1"
  4:  "1"
  5:  "2"
  6:  "3"
  7:  "4"
  8:  "5"
  9:  "Done!"
 eof  (end of output)
```

3.12 cHL: High Low Command-Line

- Status: Officially Assigned.
- Discussed: Wed, Feb 20.
- Due Date: Thu, Feb 21, 23:59
- Deadline: Thu, Mar 7, 23:59
- Grading Label: **cHL**

This is a command-line task. The general rules in section 2.3 (page 10) apply, including email subject line and program comment line.

`cis101 cHL lastname firstname` is the required subject line.

`# cis101 cHL lastname firstname` is the required comment line.

Summary: Program the high-low guessing game to run at the command line.

Task: Write a program. When it starts it should pick a number between 1 and 100 (inclusive). It should then iterate (loop) asking for a guess, and telling whether the guess is too high, too low, or correct. After the correct answer is found, it should start over, picking a new number. This will involve two infinite loops, nested.

3.13 oHL: High Low Online

- Status: Officially Assigned.
- Discussed: Fri, Feb 22, and Mon, Feb 25.
- Due Date: Mon, Mar 4, 06:00
- Deadline: Tue, Mar 12, 23:59
- Grading Label: **oHL**

This is an online task, but you do not need to send me the code that you write. It just needs to work when I test it online. The general rules in section 2.2 (page 9) apply.

If you are requesting a regrade, this is the required subject line:

```
cis101 regrade oHL lastname firstname
```

This is a two-class activity, worth double-points.

Summary: Program the high-low guessing game to run online.

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 between 1 and 100 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.

3.14 g23: Birthday

- Status: Officially Assigned.
- Discussed: Wed, Feb 27
- Due Date: Tue, Mar 5, 23:59
- Deadline: Tue, Mar 19, 23:59
- Grading Label: **g23**

This is a GradeBot task. The general rules in section 2.4 (page 11) apply, including email subject line and program comment line.

`cis101 g23 lastname firstname` is the required subject line.

`# cis101 g23 lastname firstname` is the required comment line.

Summary: Read in a date of birth. Calculate age in years. Say Happy Birthday if appropriate.

Requirements: (you must do it this way). Approved spacing and indenting is required for full credit.

Sample Execution:

```
GradeBot would have engaged your program in this dialog:
note  GBot "# debug output lines are permitted"
note  GBot "# For this program, assume today is Feb 25 2013."
  1:  "Please enter your name: " (no \n)
 in>   ......................."Michelle"
  2:  "What month (number) were you born? " (no \n)
 in>   ..................................."10"
  3:  "What day were you born? " (no \n)
 in>   ......................."25"
  4:  "What year were you born? " (no \n)
 in>   ........................"1984"
  5:  ""
  6:  "Michelle, you are 28 years old."
 eof  (end of output)
```

3.15 g71: Array

- Status: Officially Assigned.
- Discussed: Mon, Mar 4
- Due Date: Tue, Mar 5, 23:59
- Deadline: Tue, Mar 12, 23:59
- Grading Label: **g71**

This is a GradeBot task. The general rules in section 2.4 (page 11) apply, including email subject line and program comment line.

`cis101 g71 lastname firstname` is the required subject line.

`# cis101 g71 lastname firstname` is the required comment line.

Summary: Add names to an array. Report how many names were added.

Requirements: (you must do it this way). Your array must be named XYZ. You must initialize it to be empty.

You are allowed to have exactly one `<STDIN>` statement. It will be inside your main loop.

Do **NOT** tally the names as you read them in. You must use the size of the array to see how many names are in the array.

Approved spacing and indenting is required for full credit.

Sample Execution:

```
GradeBot would have engaged your program in this dialog:
note  GBot "# debug output lines are permitted"
  1:  "Name? " (no \n)
 in>   ......"Anthony"
  2:  "Name? " (no \n)
 in>   ......"Abe"
  3:  "Name? " (no \n)
 in>   ......""
  4:  "There were 2 names."
 eof  (end of output)
```

3.16 g72: Roll

- Status: Officially Assigned.
- Discussed: Wed, Mar 6
- Due Date: Thu, Mar 7, 23:59
- Deadline: Thu, Mar 14, 23:59
- Grading Label: **g72**

This is a GradeBot task. The general rules in section 2.4 (page 11) apply, including email subject line and program comment line.

`cis101 g72 lastname firstname` is the required subject line.

`# cis101 g72 lastname firstname` is the required comment line.

Summary: Add names to an array. Report how many names were added. Tell whether a specific name is in the list.

Suggestions: (you do not have to do it this way). Use a flag or counter to tell whether you found the student.

Requirements: (you must do it this way). Use push and foreach. Do not use indexing (like $x[1]). Approved spacing and indenting is required for full credit.

Sample Execution:

```
GradeBot would have engaged your program in this dialog:
note  GBot "# debug output lines are permitted"
  1:  "Who is attending? " (no \n)
 in>   .................."Wendy"
  2:  "Who is attending? " (no \n)
 in>   .................."Eve"
  3:  "Who is attending? " (no \n)
 in>   ..................""
  4:  "There are 2 students present."
  5:  "Who do you seek? " (no \n)
 in>   ................."Eve"
  6:  "Eve is present."
 eof  (end of output)
```

3.17 g73: Boring

- Status: Officially Assigned.
- Discussed: Fri, Mar 8
- Due Date: Mon, Mar 11, 06:00
- Deadline: Thu, Mar 14, 23:59
- Grading Label: **g73**

This is a GradeBot task. The general rules in section 2.4 (page 11) apply, including email subject line and program comment line.

`cis101 g73 lastname firstname` is the required subject line.

`# cis101 g73 lastname firstname` is the required comment line.

Summary: It's Dinner Time. Ask what is for dinner. If you have had it before, then it is boring. If not, then it is yummy. Repeat until you get a blank line.

Suggestions: (you do not have to do it this way). Use an outer loop to ask what is for dinner. Use an inner loop to decide whether it is boring. Use a counter or a flag to tell whether it is a repeat.

Requirements: (you must do it this way). Approved spacing and indenting is required for full credit.

Sample Execution:

```
GradeBot would have engaged your program in this dialog:
note  GBot "# debug output lines are permitted"
  1:  "What is for dinner? " (no \n)
 in>   ...................."Rice"
  2:  "Rice? That sounds yummy."
  3:  "What is for dinner? " (no \n)
 in>   ...................."Rice"
  4:  "Rice? That's boring."
  5:  "What is for dinner? " (no \n)
 in>   ...................."Noodles"
  6:  "Noodles? That sounds yummy."
  7:  "What is for dinner? " (no \n)
 in>   ...................."Rice"
  8:  "Rice? That's boring."
  9:  "What is for dinner? " (no \n)
```

```
in>    ...................""
eof  (end of output)
```

3.18 g74: Split1

- Status: Officially Assigned.
- Discussed: Mon, Mar 11.
- Due Date: Thu, Mar 14, 23:59
- Deadline: Tue, Apr 9, 23:59
- Grading Label: **g74**

This is a GradeBot task. The general rules in section 2.4 (page 11) apply, including email subject line and program comment line.

`cis101 g74 lastname firstname` is the required subject line.

`# cis101 g74 lastname firstname` is the required comment line.

Summary: Convert a date from mm/dd/yy format to dd mmm yyyy format.

Suggestions: (you do not have to do it this way). Use split to divide mm/dd/yy into parts. Use indexing to find the mmm that goes with mm. Use math to convert yy into yyyy.

Requirements: (you must do it this way). Approved spacing and indenting is required for full credit.

Sample Execution:

```
GradeBot would have engaged your program in this dialog:
note  GBot "# debug output lines are permitted"
  1:  "Date in mm/dd/yy format: " (no \n)
 in>   ........................"11/21/12"
  2:  "That is 21 Nov 2012."
 eof  (end of output)
```

3.19 oF2: Farm 2

- Status: Officially Assigned.
- Discussed: Wed, Mar 13.
- Due Date: Thu, Mar 14, 23:59
- Deadline: Tue, Apr 9, 23:59
- Grading Label: **oF2**

This is an online task, but you do not need to send me the code that you write. It just needs to work when I test it online. The general rules in section 2.2 (page 9) apply.

If you are requesting a regrade, this is the required subject line:

```
cis101 regrade oF2 lastname firstname
```

Learning Objective: Combine your skills with split, lists, and loops.

You are planting a farm. Ask for planting directions. Show the results.

Your screen should have exactly two inputs: planting directions and a submit button. Read in your directions. Use split to convert them into a list.

Make the user's directions "sticky." That is, whatever they typed should still be there after they press the submit button.

You can use the placeholder attribute to make words appear in the blank when nothing has been typed in yet.

```
<input ... placeholder=\"apples 5 guava 3\" size=100 />
```

Give sufficient directions to the user that they will probably know what to do.

While the list is not empty, shift off an item. It is the thing to be planted. Shift off an item. It is the quantity to be planted.

Print a line telling what the requested crop and quantity are. This helps in case the user types things in the wrong order. It tells the user how you are interpreting their request.

If the quantity is unreasonably large, convert it to a smaller number. Example: if quantity is more than 9, just use 9.

Print a line of that many pictures of that crop. Use the alt tag in case the user asks for a crop you do not have. You can do something like this:

```
<img src=\"$crop.jpg\" alt=\"$crop??\">
```

Repeat with the next crop until you run out of crops.

3.20 oF3: Farm 3

- Status: Officially Assigned.
- Discussed: Mon, Mar 18.
- Due Date: Tue, Mar 19, 23:59
- Deadline: Tue, Apr 9, 23:59
- Grading Label: **oF3**

This is an online task. For this one you **also** need to send me the code that you write, and it also needs to work when I test it online. The general rules in section 2.2 (page 9) apply, including email subject line and program comment line.

`cis101 oF3 lastname firstname` is the required subject line.

`# cis101 oF3 lastname firstname` is the required comment line.

Task: Same as oF1 (above) but using subroutines. You are planting a farm. Ask for planting directions. Show the results.

Your task includes writing two subroutines: `plant` and `harvest`. These subroutines will call olin as needed and do all the printing required to accomplish their tasks.

I will read your code to verify that you used the proper structure in writing your program.

3.20.1 Main Program: Crops

You must make an array (list) of the crops you are farming. This is the **only** place that the literal names of your crops may appear in the program. Everything else must be done using variables.

```
@crops = ( ... );
```

The text strings in @crops must be usable for (a) display labels, for (b) input names, and for (c) image file names.

Thus, if one of the crops 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="$fruit"` in the input field, use `olin($fruit)` to retrieve the quantity, and use `"$fruit.jpg"` to show the picture of the tomato.

3.20.2 Main Program: Planting Call

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

```
foreach $crop ( @crops ) { plant ( $crop ) }
```

Then include an appropriate "submit button."

3.20.3 Subroutine "Plant"

Do not use any global variables.

Write a subroutine named `plant` that does the following:

Display the name of the crop to be planted. (The name of the crop comes from the subroutine's parameter list.)

Display a blank into which a number can be entered. Make the number "sticky." That is, whatever they typed should still be there when the screen is redrawn after they press the submit button. (The quantity comes from a call to `olin`.)

3.20.4 Main Program: Harvesting Call

Use the following foreach loop to display the harvest.

```
foreach $crop ( @crops ) { harvest ( $crop ) }
```

3.20.5 Subroutine "Harvest"

Do not use any global variables.

Print a line telling what the requested crop and quantity are. (The name of the crop comes from the subroutine's parameter list. The quantity comes from a call to `olin`.)

After printing it, if the quantity is unreasonably large, convert it to a smaller number. Example: if quantity is more than 9, just use 9. At your discretion, complain about the size of the number.

Print a row of that many pictures of that crop. For example:

```
<img src=\"$crop.jpg\" alt=\"$crop\">
```

3.21 oLT: LocalTime

- Status: Officially Assigned.
- Discussed: Fri, Mar 22.
- Due Date: Mon, Mar 25, 06:30
- Deadline: Tue, Apr 9, 23:59
- Grading Label: **oLT**

This is an online task, but you do not need to send me the code that you write. It just needs to work when I test it online. The general rules in section 2.2 (page 9) apply.

If you are requesting a regrade, this is the required subject line:

```
cis101 regrade oLT lastname firstname
```

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, with month as a word, not as a number. If you are really good, add other things, such as the day of the week.

You should be able to do this without using any “if” statements.

3.22 g43: Leap Year

- Status: Officially Assigned.
- Discussed: Mon, Mar 25.
- Due Date: Tue, Mar 26, 23:59
- Deadline: Tue, Apr 9, 23:59
- Grading Label: **g43**

This is a GradeBot task. The general rules in section 2.4 (page 11) apply, including email subject line and program comment line.

`cis101 g43 lastname firstname` is the required subject line.

`# cis101 g43 lastname firstname` is the required comment line.

Summary: Tell whether a particular year is a leap year or not.

```
  If a year has Feb 29, then it is a leap year.
  Tell whether a year is leap year or not.
  If the year is a multiple of 4, then it is.
  But if it is a multiple of 100, then it is not.
  But if it is a multiple of 400, then it is.
```

Sample Execution:

```
GradeBot would have engaged your program in this dialog:
note  GBot "# debug output lines are permitted"
  1:  "What is the year? " (no \n)
 in>   .................."2011"
  2:  "2011 is not a leap year."
 eof  (end of output)
```

3.23 g75: Split2: Tally

- Status: Officially Assigned.
- Discussed: Wed, Mar 27.
- Due Date: Thu, Mar 28, 23:59
- Deadline: Tue, Apr 9, 23:59
- Grading Label: **g75**

This is a GradeBot task. The general rules in section 2.4 (page 11) apply, including email subject line and program comment line.

`cis101 g75 lastname firstname` is the required subject line.

`# cis101 g75 lastname firstname` is the required comment line.

Summary: Add up a line of numbers.

Suggestions: (you do not have to do it this way). Use split and foreach.

Requirements: (you must do it this way). Approved spacing and indenting is required for full credit.

Sample Execution:

```
GradeBot would have engaged your program in this dialog:
note  GBot "# debug output lines are permitted"
  1:  "Numbers: " (no \n)
 in>   ........."1 3 5 6"
  2:  "The total is 15."
 eof  (end of output)
```

3.24 g44: Phonecard

- Status: Officially Assigned.
- Discussed: Mon, Apr 1.
- Due Date: Tue, Apr 2, 23:59
- Deadline: Tue, Apr 9, 23:59
- Grading Label: **g44**

This is a GradeBot task. The general rules in section 2.4 (page 11) apply, including email subject line and program comment line.

`cis101 g44 lastname firstname` is the required subject line.

`# cis101 g44 lastname firstname` is the required comment line.

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

If the formatted savings is zero, 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"
  $savings = sprintf ( "%.2f", $savings );
  print "blah blah $savings blah blah\n";
```

Sample Execution:

```
GradeBot would have engaged your program in this dialog:
note  GBot "# debug output lines are permitted"
  1:  "Welcome to the Prepaid Phone Card Analysis Program"
  2:  ""
  3:  "Tell me about your phone cards and your calling habits."
  4:  "I'll tell you the best card to use."
  5:  ""
  6:  "For the first card"
  7:  "  Enter connect cost per call: $" (no \n)
```

```
in>   ..............................."0"
 8:  "  Enter additional cost per minute: $" (no \n)
in>   ...................................".07"
 9:  "For the second card"
10:  "  Enter connect cost per call: $" (no \n)
in>   ...............................".49"
11:  "  Enter additional cost per minute: $" (no \n)
in>   ...................................".03"
12:  "Enter estimated call duration: " (no \n)
in>   ..............................."5"
13:  "You would save $0.29 by using the first card."
14:  ""
15:  "Thank you for using the Prepaid Phone Card Analysis Program."
eof  (end of output)
```

3.25 g45: Afford

- Status: planned
- Discussed: Wed, Apr 3
- Due Date: Thu, Apr 4, 23:59
- Deadline: Tue, Apr 9, 23:59
- Grading Label: **g45**

This is a GradeBot task. The general rules in section 2.4 (page 11) apply, including email subject line and program comment line.

`cis101 g45 lastname firstname` is the required subject line.

`# cis101 g45 lastname firstname` is the required comment line.

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.

Sample Execution:

```
GradeBot would have engaged your program in this dialog:
note  GBot "# debug output lines are permitted"
  1:  "What is the price of item 1? " (no \n)
 in>   ............................"1"
  2:  "What is the price of item 2? " (no \n)
 in>   ............................"2"
  3:  "How much money do you have? " (no \n)
 in>   ..........................."3"
  4:  "Buy both!"
 eof  (end of output)
```

3.26 oJS: JavaScript

- Status: planned
- Discussed: Fri, Apr 5
- Due Date: Mon, Apr 8, 00:01
- Deadline: Tue, Apr 9, 23:59
- Grading Label: **oJS**

This is an online task, but you do not need to send me the code that you write. It just needs to work when I test it online. The general rules in section 2.2 (page 9) apply.

If you are requesting a regrade, this is the required subject line:

```
cis101 regrade oJS lastname firstname
```

Task: Create a web page (a) properly linked to the student projects page. It should include (b) your name, (c) a JavaScript calculator similar to the one on my demo page.

My calculator has buttons for add, subtract, multiply, and divide.

Go beyond the example.

The divide button fails when you divide by zero. See if you can figure out how to fix it.

Consider adding other buttons, like maybe square root, or $a^2 + b^2$, or something totally random.

Chapter 4

Activities Pending

Contents

This is a list of the activities that are being considered and may be assigned. Or maybe they will not be assigned. When they are actually assigned, they will be moved from this chapter into the "Assigned" chapter.

You are welcome to work on these before they are assigned, but please do not submit them until they are actually assigned.

4.1 g62: For Loop

- Status: planned
- Discussed:
- Due Date:
- Deadline: Tue, Apr 9, 23:59
- Grading Label: **g62**

This is a GradeBot task. The general rules in section 2.4 (page 11) apply, including email subject line and program comment line.

`cis101 g62 lastname firstname` is the required subject line.

`# cis101 g62 lastname firstname` is the required comment line.

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.

Sample Execution:

```
GradeBot would have engaged your program in this dialog:
note  GBot "# debug output lines are permitted"
  1:  "Where should I start? " (no \n)
 in>   ......................"1"
  2:  "Where should I end? " (no \n)
 in>   ...................."5"
  3:  "What should I count by? " (no \n)
 in>   ........................"1"
  4:  "1"
  5:  "2"
  6:  "3"
  7:  "4"
  8:  "5"
  9:  "Done!"
 eof  (end of output)
```

Chapter 5

GradeBot

Contents

`http://gradebot.tk/` is the web interface for what used to be a huge system called GradeBot. What you see at that URL is called GradeBot Lite.

`http://gbot.dc.is2.byuh.edu/` is an alternate URL in case the tk URL stops working.

GradeBot is an automated program grader.

5.1 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.
```

5.2 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.

5.3 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.

Follow the email rules in section 2.1 (page 7) as you construct your subject line and the body of your message.

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 6

Style Requirements

Contents

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.

6.1 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.

6.2 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`

6.3 Parentheses: Math vs Array

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.

6.4 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.

6.5 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.

6.6 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.

6.7 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 may be used in short-range contexts where their meaning is clear by the immediately surrounding code. Like "he", "she", and "it" in English, they become confusing in wider contexts. Use something more meaningful.

Chapter 7

Exam Questions

Contents

In this chapter, we consider each exam question. We identify the key things you need to demonstrate. We mention the common mistakes that people make. Before attempting a problem (either for the first time or a subsequent time), you might benefit from reviewing the section that talks about that problem.

7.1 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.

7.2 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.

7.3 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.

7.4 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.

7.5 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.

7.6 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.

7.7 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 (and when) to handle all possible capitalizations. What does “dictionary order” mean?

(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.

7.8 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.

7.9 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.

7.10 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.

7.11 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.

7.12 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";`

7.13 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 $_ }`

7.14 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.

7.15 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] }`

7.16 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)

7.17 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)

7.18 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.

7.19 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.

7.20 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.

7.21 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`.

7.22 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 8

Final Projects

- Status: Officially Assigned.
- Discussed:
- Due Date: Tue, Apr 9, 24:00
- Deadline: Tue, Apr 9, 24:00

The final project is due by midnight on Tuesday night, the day after the last day of class. I plan to grade it early on Wednesday unless you have asked me to grade yours earlier.

(50) Project Points
10 Project CGI: write a dynamic web page
10 Project Pictures: use img tags
15 Project Multi Input: process multiple inputs
15 Project Hidden Fields: pass state (counter, etc)

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

Doing a project is a great way for you 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 want or 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.

How To Submit It

Final projects must be linked to my student projects page, proj column, which links to your /proj/ directory.

I will automatically check for all projects soon after the deadline, so you do not need to tell me that you are doing a project or not.

However, if you want me to review and possibly grade your final project early, you can send me an email. Use the following subject line, or something very close to it.

Subject Line: `cis101 final project, lastname, firstname`

It will speed things up for both of us if you could please include a clickable link to your project right in your email. It is likely to get you a faster response.

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.

Invent a Project

Doing pre-defined projects is often 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: For any credit at all, your program must run online as a web application. Anyone in the world should be able to run your program. This part is absolutely required.

Authorship: The code comments and the program output (webpages) 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: For maximum credit, your program must 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.

Multi Input: For maximum credit, 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. All your submit buttons except the first count as inputs if they each do something different. Your first submit button does not count.

State: For maximum credit, 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. It could be a counter or a running total or anything else that is "state." And hidden fields count towards the multiple input requirement.

Index