CIS 101 Study Guide
Summer 2015

Don Colton
Brigham Young University–Hawai‘i

June 10, 2015

Study Guide

This is the official study guide for the CIS 101 class, Beginning Programming, as taught by Don Colton, Summer 2015. It is focused directly on the grading of the course.

`http://byuh.doncolton.com/cis101/2153/sguide.pdf` is the study guide, which is this present document. **It will be updated frequently throughout the semester,** as new assignments are made, and as due dates are established, and as clarifications are developed.

Syllabus

`http://byuh.doncolton.com/cis101/2153/syl.pdf` is the official syllabus for this course. It is largely reproduced in Chapter 1 (page 3) below.

Textbook

This study guide is a companion to the textbook for the class, Introduction to Programming Using Perl and CGI, Third Edition, by Don Colton.

The textbook is available here, in PDF form, free.

`http://ipup.doncolton.com/`

The textbook provides explanations and understanding about the content of the course.

Contents

Chapter 1

Syllabus

The original, separate syllabus is the official version. A copy of it is included here for (a) convenience, (b) as a place to correct minor errors that may be in the original, and (c) because this study guide has an index but the syllabus alone does not.

Contents

1.1 Overview

Computers are great. But they are also really stupid.

By stupid, I mean computers only understand really simple commands. Anything complex must be built up out of these simple commands.

Programming is the art of building up the fun and interesting things that you want to be done, starting from just the really simple commands that the computer can understand.

Sometimes it is frustrating. Sometimes it is very satisfying.

This class teaches skills by which you can better serve those around you. It teaches skills you can "take to the bank."

There are many fine programming languages. Our programming language will be Perl.

1.1.1 General Education Breadth

As of 2015, CIS 101 is listed among the BYUH General Education requirements as an approved Breadth of Knowledge option in the Science and Technology category. The classes in that category include: ASTR 104, BIOL 100/112, CHEM 100/105, GEOL 105, PHSC 100, PHYS 100, PHYS 121, CIS 100, CIS 101, and IT 240.

The Breadth of Knowledge categories are intended to give people credit for something they are already taking in their major area, and give them a chance to explore a broader range of fields.

Before 2015, most students took this class because it is a major requirement. Moving forward, many students are taking it to explore an area of interest. Starting Fall 2014 I have been adjusting the structure and content of the course to serve all students better. We no longer assume, for example, that everyone wants to become an expert. If you have suggestions, please let me know.

1.1.2 Expected Proficiencies

As you begin this course, we assume you have no programming experience whatsoever. We expect you can read, type, send and receive email, and visit web sites. We will teach you everything else you need to know.

Ideally you will have your own personal computer, probably a laptop, on which you can write and test programs. (You can also use the computers in the CIS department labs.) The textbook tells how to install Perl on a Windows machine, and how to find it (pre-installed) on a Macintosh.

1.1.3 Retaking CIS 101

If you are retaking, do not reuse your old work. That would hurt your learning. Do not even look at any of your old work until the course is ended. Start fresh and redo everything as though you were taking the class for the first time.

1.2 Course and Faculty

1.2.1 Course Information

- **Title:** Beginning Programming
- **Course Number:** CIS 101
- **Course Description:** (from the catalog) Extensive hands-on software development and testing using variables, arrays, instruction sequences, decisions, loops, and subroutines. May also include dynamic web pages (CGI) and regular expressions.
- **Prerequisites:** none
- **Semester/Year:** Summer 2015
- **Semester Code:** 2153
- **Meeting Time:** MWF 12:10 to 14:20
- **Location:** GCB 111
- **First Day of Instruction:** Mon, Apr 27
- **Last Day to Withdraw:** Thu, May 28
- **Last Day for Late Work:** Wed, Jun 10
- **Last Day of Instruction:** Wed, Jun 10
- **Final Exam:** Fri, Jun 12, 12:10 to 14:20

1.2.2 Faculty Information

- **Instructor:** Don Colton
- **Office Location:** GCB 128
- **Office Hours:** MWF 11:00-12:00.
- **Email:** doncolton2@gmail.com
- **Campus Homepage:**
  http://byuh.doncolton.com/ is my campus homepage. It has my calendar and links to the homepages for each course I teach.
- **Off-Campus Homepage:**
  http://doncolton.com/ is my off-campus homepage.

There is no special lab hour this semester.

1.2.3 Course Readings and Materials

- **Textbook:**
  http://ipup.doncolton.com/ Introduction to Programming Using

Perl and CGI, by Don Colton.

- **Learning Management System:**
  https://dcquiz.byuh.edu/ is the learning management system for my courses.
- **Course Homepage:**
  http://byuh.doncolton.com/cis101/ is my course homepage. It has links to many things including the syllabus, study guide, and textbook.
- **Study Guide:**
  http://byuh.doncolton.com/cis101/2153/sguide.pdf is the study guide for this course. It includes a copy of most parts of this syllabus. The study guide is updated frequently throughout the semester as assignments are made and deadlines are established or updated.

1.3 Calendar

Mon	Apr	27	o1H, g21
Wed	Apr	29	g33, o1R, Read Unit 1: Output
Fri	May	01	g35, o1D, Read Unit 2: Input
Mon	May	04	g41, c2M, Read Unit 3: Math
Wed	May	06	Exam v0 prac (short day)
Fri	May	08	g42, o2M, Read Unit 4: Decisions
Mon	May	11	g51, g61, Read Unit 5: Decisions
Wed	May	13	o6F, Read Unit 6: Loops
Fri	May	15	g46, **Exam v1**
Mon	May	18	g47, o6D
Wed	May	20	g73, o6M, Read Unit 7: Arrays
Fri	May	22	**Exam v2**
Mon	May	25	Memorial Day No Class
Wed	May	27	g45, o7T
Thu	May	28	Last Day To Withdraw
Fri	May	29	**Exam v3**
Mon	Jun	01	g78, o7F, Read Unit 8: Subroutines
Wed	Jun	03	g65, o6H
Fri	Jun	05	**Exam v4**
Mon	Jun	08	g43, g7A
Wed	Jun	10	oJS, Last day for late work
Fri	Jun	12	**Exam v5**, 12:10 to 14:20, GCB 111

We meet 20 times including the final exam.

Exam dates are firm and will not change unless there is a massive emergency. Exams are closed-book, closed-notes, closed-neighbor, etc. You can bring blank paper.

1.4 Grading

I use a 60/70/80/90 model based on 1000 points.

Based on 1000 points

930+	A	900+	A–	870+	B+
830+	B	800+	B–	770+	C+
730+	C	700+	C–	670+	D+
630+	D	600+	D–	0+	F

The 1000 points are divided up as follows.

Daily Updates 20x2 points.

Daily Quizzes 135 points.

Activities 300 points.

Exams 525 points.

If you want to major in CS, IS, or IT, you need to earn a C or better (730 points or more) in this class.

1.4.1 CIS 101 Grade Books

In my Learning Management System (DCQuiz), I keep several online grade books so you can see how your points are coming along. This lets you compare yourself with other students in the class (without seeing their names).

2153 CIS 101 Overall Grade Book: This includes the totals from all the other grade books. This is where you can find your final grade at the end of the course.

2153 CIS 101 (whatever) Grade Book shows your points in the (whatever) category. (whatever) is Attendance, Daily Quiz, Activities, or Exam.

1.4.2 Attendance (20x2 points)

Each day in class starts with the "daily update" (DU). It is my way of reminding you of due dates and deadlines, sharing updates and news, and taking roll. It is your way of saying something anonymously to each other and to me. It must be taken in class at a classroom computer during a window of time that starts a few minutes before class and ends 5 minutes into class.

Attendance: You must attend to earn the Attendance points. You must attend to earn the Daily Quiz points. You must attend to earn the exam points. Besides that, there is no penalty for being late or lack of attendance. You can do the daily activities without attending, for instance.

2153 CIS 101 Attendance Grade Book shows your attendance points, two points per day, for 20 days. You get two points for each time you do the daily update. If you arrive too late to complete the daily update, you will not receive the attendance points for that day.

Tardiness: My tardiness policy is that you should arrive in time to complete the daily update. Generally if you are less than four minutes late, you will have time to complete the daily update before the deadline.

1.4.3 Daily Quiz (135 points)

There are assigned readings. These are listed in the course calendar. Right after prayer on many days (but not exam days), there will be a short quiz. It will consist of several randomly chosen questions from the assigned readings.

The daily quizzes are "closed book," by which I mean that you are not allowed to look up answers while you are taking the quiz.

Your scores from these Daily Quizzes will be recorded in **2153 CIS 101 Daily Quiz Grade Book**. The total from this grade book will be rescaled so the top score is worth 135 points.

1.4.4 Daily Activities (300 points)

On most days we will have an in-class activity assignment. Each will normally be worth 10 points. Roughly 30 assignments x 10 points = 300 points.

2153 CIS 101 Activities Grade Book tracks your performance on daily activities. The number of in-class activities is not perfectly predictable. The total from this grade book will be rescaled so the full-credit values add up to 300.

Assignments are correct when they work properly and comply with all other stated requirements (such as style and algorithm). Points are assigned according to the date on which the correct work is received. Details are provided in the study guide, but in general it works like this:

11: Correct by 23:59 the same class day it was discussed or assigned. This includes a ten percent bonus for turning it in early.

10: Correct by 23:59 of the next class day. This is full credit.

7: Correct by 23:59 of Wed, Jun 10, the last day for late work. This is partial credit for being late.

1.4.5 Do Your Own Work

Help each other, but do your own work.

These two goals are in conflict with each other. To resolve this conflict, I draw the line at copying.

Except during exams, you are strongly encouraged to work with your fellow students. We want everyone to get full credit on every assignment. Please help each other learn. That is the goal.

If You are Looking at Someone's Program:

You **may**: look, read, make mental notes, ask questions, point out possible errors, and try to understand.

You **must not**: fix someone else's program, take written notes, take pictures, take a copy of all or part of their program.

If Someone is Looking at Your Program:

You **may**: explain your approach, ask for help.

You **must not**: let them fix your program, give them a copy of all or part of your program.

This includes working with tutors or students who took the class in previous semesters.

1.4.6 Final Exams (525 points)

You get several chances to pass the final exam. There are 21 sections to the exam. They are scored and tracked separately. You should pass each section at least once.

Final Exam Dates:

Wed May 06	Exam v0 (Practice)
Fri May 15	Exam v1, 13:20 to 14:20, GCB 111
Fri May 22	Exam v2, 12:10 to 14:20, GCB 111
Fri May 29	Exam v3, 12:10 to 14:20, GCB 111
Fri Jun 05	Exam v4, 12:10 to 14:20, GCB 111
Fri Jun 12	Exam v5, 12:10 to 14:20, GCB 111

There are 21 exam tasks. Each is a program for you to do during one of the final exams. Each is worth 25 points. Points for each question can be earned only once.

Each exam is a "final exam" in the sense that it covers everything we learn during the semester, and by completing it, you earn the points for it as though you had done it on the day of the actual final. One practice exam is also given, for no credit, to help you understand how to do the other tests.

(525) Exam Points (21 tasks)

1 25p String Basic

2 25p Number Basic

3 25p Number Story

4 25p Number Decision

5 25p Number Decision Story

6 25p String Decision

7 25p String Decision Bracket

8 25p Repeat While

9 25p Repeat For

10 25p Repeat Last

11 25p Repeat Nested

12 25p Lists Basic

13 25p Lists Loop

14 25p Arrays Basic

15 25p Arrays Loop

16 25p Split

17 25p Join

18 25p Subroutine Returns

19 25p Subroutine Positional Parameters

20 25p Subroutine Globals and Locals

21 25p Subroutine Variable Parameters

The study guide talks more about each of these tasks.

1.4.7 Other Extra Credit

Extra credit is available for reporting an error in my formal communications (the published materials I provide), so I can fix it. In this class, the materials include the following:

- The course website, parts relating to this semester.
- The course syllabus.
- The course study guide.
- The course textbook, since I wrote it.

Each error reported can earn you extra credit. (Typos in my email messages are all too common and do not count.)

Syllabus errors (unless they are major) will probably be fixed only in the study guide. Check there before reporting it.

1.5 Instructional Methods

Exams happen on scheduled exam days. This instructional method brings you face to face with the challenges you need to be able to solve.

Lectures happen occasionally. I review material that was assigned from the text book and do what I can to make it clear and interesting. These can be short or take up most of the class hour. Longer ones happen more often at the start of the course than they do later on.

Activity days are usually the most common. A learning activity is assigned. Typically it is a program to be written. The program will be described in the study guide. I often give an overview of the problem and the techniques that I think will be helpful to solve it. Typically this takes about 15 minutes, but the actual time varies widely. Then I sit down at the front of the room and invite students to visit with me, one on one, for assistance. Students are also encouraged to help each other.

Help in Class: As students come to visit with me, I call up their computer screen from the place they were sitting, and we look at their program code or whatever else the student is asking about. We review the situation together. The student then returns to work on their program at their seat and I work with the next student waiting in line.

I want to help as many students as I can. You can help by doing these kinds of things before coming up.

(a) If asking about an assignment, have my study guide available in a tab on your browser, turned to the relevant assignment so we can review the requirements.

(b) If asking about grades, have my grade book available.

(c) If asking about an exam question, have that exam available.

1.5.1 BYUH Learning Framework

I agree with the BYUH Framework for Learning. If we follow it, class will be better for everyone.

Prepare for CIS 101

Prepare: Before class, study the course material and develop a solid understanding of it. Try to construct an understanding of the big picture and how each of the ideas and concepts relate to each other. Where appropriate use study groups to improve your and others' understanding of the material.

In CIS 101: Make reading part of your study. There is more than we could cover in class because we all learn at different rates. Our in-class time is better spent doing activities and answering your questions than you listening to a general lecture.

Engage in CIS 101

Engage: When attending class actively participate in discussions and ask questions. Test your ideas out with others and be open to their ideas and insights as well. As you leave class ask yourself, “Was class better because I was there today?”

In CIS 101: Participate in the in-class activities. As you make progress, assist those around you that want assistance. It is wonderful what you can learn by trying to help someone else.

Improve at CIS 101

Improve: Reflect on learning experiences and allow them to shape you into a more complete person: be willing to change your position or perspective on a certain subject. Take new risks and seek further opportunities to learn.

In CIS 101: After each exam, I normally allow you to see every answer submitted, every score given, and every comment I wrote, for every question, until the next exam happens. Compare your answers to those of other students. See how your answers could be improved. If you feel lost, study the readings again or ask for help.

1.5.2 Support

The major forms of support are (a) open lab, (b) study groups, and (c) tutoring.

If you still need help, please find me, even outside my posted office hours.

Office Hour / Open Lab

Office hours are MWF 11:00-12:00.

Study Groups

You are encouraged to form a study group. If you are smart, being in a study group will give you the opportunity to assist others. By assisting others you will be exposed to ideas and approaches (and errors) that you might never have considered on your own. You will benefit.

See section 1.4.5 (page 10) for some rules.

If you are struggling, being in a study group will give you the opportunity to ask questions from someone that remembers what it is like to be totally new at this subject. They are more likely to understand your questions because they sat through the same classes you did, took the same tests as you did, and probably thought about the same questions that you did.

Most of us are smart some of the time, and struggling some of the time. Study groups are good.

Tutoring

The CIS department provides tutoring in GCB 111, Monday through Friday, typically starting around 17:00 and ending around 23:00 (but earlier on Fridays). Normally a schedule is posted on one of the doors of GCB 111.

Tutors can be identified by the red vests they wear when they are on duty.

The best way to work with a tutor is to show them something that you have written and ask them why it does not work the way you want. This can open the door to a helpful conversation.

Another good way to work with a tutor is to show them something in the textbook and ask about it.

Please do not plunk down next to a tutor and say, “I don’t understand. Can you teach me?” If you did not try hard to read carefully, you are wasting everybody’s time.

1.6 Course Policies

Subject to Change: Like all courses I teach, I will be keeping an eye out for ways this one could be improved. Changes generally take the form of opportunities for extra credit, so nobody gets hurt and some people may be helped. If I make a change to the course and it seems unfair to you, let me know and I will try to correct it. If you still think it is unfair, you can appeal to the department chair or the dean. Also, you are welcome to suggest ways you think the class could be improved.

Digital Recording by me: I may digitally record the audio of my lectures some days. This is to help me improve my teaching materials.

Digital Recording by you: Almost everyone has a smart phone these days. I assume students will freely record what goes on in class, and take pictures of what is on the white board, to aid in their studies. I simply ask that you not embarrass anyone.

1.6.1 Special Treatment

There are many good reasons why students request special treatment. These include, for example, illness, field trips, performances, athletic events, and special needs. Instead of dealing with these as they arise, based on my past experience, I have adopted general policies that are intended to accommodate all but the most difficult cases, and thereby avoid the need for special treatment.

The exams are virtually identical, and given many times. If you miss one because of travels or illness or any other reason, just make it up by doing well on the next test. If you must miss three or more exams, you may have a special case and should see me.

1.6.2 Reasonable Accommodation

This section covers special needs, including qualified special needs, as well as all other requests for special treatment.

I have carefully designed each of my classes to provide reasonable accommodation to those with special needs. Beyond that, further accommodation is usually considered to be unreasonable and only happens in extreme cases. Please see the paragraph on “Accommodating Special Needs” below for more information.

Ample Time: Specifically, I allow ample time on tests so that a well-prepared student can typically finish each test in half of the time allowed. This gives everyone essentially double the amount of time that should normally be needed.

Exam Retakes: There are no make-up exams. I give the final exam a number of times and some students are able to complete it before the last time.

Deadlines: Most assignments are due a few days after they are discussed, but I normally allow late work up until 23:59 on Wed, Jun 10. See the study guide for specific deadlines.

Even though I truly believe that these methods provide reasonable accommodation for almost everyone in almost every case, you might have a highly unusual situation for which I can and should do even more. You are welcome to see me about your situation.

1.6.3 Communication

We communicate with each other both formally and informally.

Formal communication is official, carefully worded, and normally in writing. Formal is for anything truly important, like grades.

Informal communication is casual and impromptu. It is meant to be helpful and efficient. Reminders are informal. Emails are informal. Explanations are usually informal.

From Me to You, Formal

I communicate formally, in writing, through (a) the syllabus, (b) the study guide, and (c) the learning management system.

(a) Syllabus: `http://byuh.doncolton.com/cis101/2153/syl.pdf` is the syllabus for this course. It tells our learning objectives and how you will be graded overall. You can rely on the syllabus. After class begins, it is almost never changed except to fix major errors.

(b) Study Guide: `http://byuh.doncolton.com/cis101/2153/sguide.pdf` is the study guide for this course. It includes a copy of the syllabus. The study guide is updated frequently throughout the semester, as assignments are made and deadlines are established or updated.

(b1) Calendar: The study guide tells when things will happen. It contains specific due dates.

(b2) Assignments: The study guide tells what assignments have been made and how you will be graded, item by item. It provides current details and specific helps for each assignment. It provides guidance for taking the exams.

(c) DCQuiz: `https://dcquiz.byuh.edu/` is my learning management system. I use it to give tests. I use it to show you my grade books.

From Me to You, Informal

My main informal channels to you are (a) word of mouth and (b) email.

(a) Word of Mouth, including Lecture: Class time is meant to be informative and helpful. But if I say anything truly crucial, I will also put it into the study guide.

(b) Email: My emails to you are meant to be helpful. But if I say anything truly crucial, I will also put it into the study guide. Normally I put CIS 101 at the front of the subject line in each email I send.

From You to Me, Formal

Your formal channels to me, specifically how you turn in class work, are mainly via (a) the learning management system, (b) email, and (c) specifically requested projects.

(a) DCQuiz: To use my learning management system, you must log into it. Then, you can respond to questions I have posted. Each day there will be a “Daily Update.” Exams will also be given using DCQuiz.

(b) Email: You will use formal email messages to submit some of the programs you write and to tell me certain other things. The study guide tells how to send formal emails, including where to send them, what subject line to use, and what to put in the body of the message.

(c) Student Projects: The study guide may tell you to submit certain work in the form of a webpage or web-based program. If so, it will say specifically where to put it. I will go to that spot to grade it.

From You to Me, Informal

Your informal channels to me, typically how you ask questions and get assistance, are mainly face to face and by email or chat.

Face to Face: If you need help with your class work, I am happy to look at it and offer assistance. Often this happens during class or during office hours. Often I will have you put your work on your computer screen, and then I will take a look at it while we talk face to face.

Email / Chat: You can also get assistance by sending me an email or doing a chat. I will do my best to respond to it in a reasonable and helpful way. If you want something formal, use the formal rules.

If you are writing about several different things you will usually get a faster response if you break it up into several smaller emails instead of one big email. I try to respond to a whole email at once, and not just part of it. I usually answer smaller and simpler emails faster than big ones.

1.7 Learning Outcomes

Outcomes (sometimes called objectives) are stated at several levels: Institutional (ILO), Program (PLO), and Course (CLO). In this section we set forward these outcomes and tell how they are aligned with one another.

1.7.1 ILOs: Institutional Outcomes

ILO: Institutional Learning Outcomes (ILOs) summarize the goals and outcomes for all graduates of BYUH.

Brigham Young University Institutional Learning Objectives (ILOs) Revised 24 February 2014

Graduates of Brigham Young University–Hawai‘i will:

Knowledge: Have a breadth of knowledge typically gained through general education and religious educations, and will have a depth of knowledge in their particular discipline.

Inquiry: Demonstrate information literacy and critical thinking to understand, use, and evaluate evidence and sources.

Analysis: Use critical thinking to analyze arguments, solve problems, and reason quantitatively.

Communication: Communicate effectively in both written and oral form, with integrity, good logic, and appropriate evidence.

Integrity: Integrate spiritual and secular learning and behave ethically.

Stewardship: Use knowledge, reasoning, and research to take responsibility for and make wise decisions about the use of resources.

Service: Use knowledge, reasoning, and research to solve problems and serve others.

1.7.2 PLOs: Program Outcomes

PLO: Program Learning Outcomes (PLOs) summarize the goals and outcomes for graduates in programs for which this course is a requirement or an elective. These support the ILOs, but are more specific.

At the end of this section, we include the relevant page from the CIS Program Outcomes Matrix, dated April 2011.

The following outcomes are pursued at the "Introduced" level, and apply to one or more of the majors that use this course.

(a) An ability to apply knowledge of computing and mathematics appropriate to the discipline.

(b) An ability to analyze a problem, and identify and define the computing requirements appropriate to its solution.

(i) An ability to use current techniques, skills, and tools necessary for computing practice.

(CS j) An ability to apply mathematical foundations, algorithmic principles, and computer science theory in the modeling and design of computer-based systems in a way that demonstrates comprehension of the trade-offs involved in design choices.

(CS k) An ability to apply design and development principles in the construction of software systems of varying complexity.

CIS Department Outcomes Matrix, April 2011

Program Outcomes

(a) An ability to apply knowledge of computing and mathematics appropriate to the discipline.
(b) An ability to analyze a problem, and identify and define the computing requirements appropriate to its solution.
(c) An ability to design, implement, and evaluate a computer-based system, process, component, or program to meet desired needs.
(d) An ability to function effectively on teams to accomplish a common goal.
(e) An understanding of professional, ethical, legal, security and social issues and responsibilities.
(f) An ability to communicate effectively with a range of audiences.
(g) An ability to analyze the local and global impact of computing on individuals, organizations, and society.
(h) Recognition of the need for and an ability to engage in continuing professional development.
(i) An ability to use current techniques, skills, and tools necessary for computing practice.

CS Only

(j) An ability to apply mathematical foundations, algorithmic principles, and computer science theory in the modeling and design of computer-based systems in a way that demonstrates comprehension of the tradeoffs involved in design choices. [CS]
(k) An ability to apply design and development principles in the construction of software systems of varying complexity. [CS]

IS Only

(j) An understanding of processes that support the delivery and management of information systems within a specific application environment. [IS]

IT Only

(j) An ability to use and apply current technical concepts and practices in the core information technologies. [IT]
(k) An ability to identify and analyze user needs and take them into account in the selection, creation, evaluation and administration of computer-based systems. [IT]
(l) An ability to effectively integrate IT-based solutions into the user environment. [IT]
(m) An understanding of best practices and standards and their application. [IT]
(n) An ability to assist in the creation of an effective project plan. [IT]

R = Required in that program | **CSS** = CS B.S. |**CIS** = CIS B.S. | **IS** = IS B.S. | **IT** = IT B.S.
# = choose at least 9 cr hrs | O = optional as a substitute | L = Introduced, M = Practiced with feedback, H = Demonstrated at the Mastery level

Course	**Description**	**CSS**	**CIS**	**IS**	**IT**	**a**	**b**	**c**	**d**	**e**	**f**	**g**	**h**	**i**	**CSj**	**CSk**	**ISj**	**ITj**	**ITk**	**ITl**	**ITm**	**ITn**
CIS 100	Fundamentals of Info. Systems & Tech.			R	R	L	L	L	L	L	L	L	L	L			L	L	L			
CIS 101	Beginning Programming	R	R	R	R	L	L							L	L	L						
CIS 202	Object-Oriented Programming	R	R	R	R	M	M	M		L			L	M	L	L		M	L		L	L
CIS 205	Discrete Mathematics I	R	R	R	R	M	M	L	L					M	M	M						
CIS 206	Discrete Mathematics II	R	R	R		M	M	L	L					M	M	M						
CIS 305	Systems Engineering I	R	R	R	R	M	M	M	M	L	L	M	L	M	L	L	M	L	H	L	H	M
CIS 401	Web Application Development	R		R	R	M	L	L						M			L	M	L	L		
CIS 405	Systems Engineering II	R	R	R	R	M	M	M	M	L	M	M	M	M	M	M	M	M	H	M	H	M
CIS 470	Ethics in Computer & Info. Sciences	R	R	R	R		L	L	M	H	H	H	H									
CS 203	Object-Oriented Programming II	R				M	M	M						M	M	M						
CS 210	Computer Organization	R			R	H	M	L							M	L		M				
CS 301	Algorithms & Complexity	R				L	M	L	L		M		L	M	H							
CS 320	Computational Theory	R				H	M			L		L	M		H	M						
CS 415	Operating Systems Design	R				H	H	H		M	M	M	H	H	H	H					M	
CS 420	Programming Languages	R				H	H	H		M	M	M	H	H	H	H						
CS 490R	Adv Topics in Computer Science (6 CR)	R				H	H	H					H		H	H						
IS 330	Management Information Systems					L	L		M	L	M	L	L	L			L					
IS 350	Database Management Systems	R	R	R	R	M	L	M	M	L	L	L	L	M	M	L	L	H	L			
IS 430	ITS – Enterprise Resource Planning			R			L	M	M	M	M	M	M	H			H		L		M	
IS 435	Advanced Concepts ERP Systems					H	H		H	L	M	M	M	H			H			L	H	
IS 485	Project Management & Practice			R		M	H	M	H	M	H	M	H	M	M	H	H	M				H
IT 220	Linux Essentials				R	M								M				M				
IT 224	Computer Hardware & Systems Software			R	R	M	H	L	M	L	M	L	L	L				M	M	L	L	
IT 240	Fund. Of Web Design & Technology			R	R	L	L	L		M	H	M		M		L	L	M	M	M	L	
IT 280	Data Comm. Systems & Networks	R	R	R	R	M	M	M		M	M		L	M				M	L	L		
IT 420	Linux System Administration				R	H	H	M						H				M	M	M		
IT 426	Computer Network Services				R	H	H	M	L	L	L	L	L	M				H	M	M	M	L
IT 440	Foundations of HCI				R	M	H	H	M	H	M	H	M	M			H	M	H	H	H	M
IT 480	Computer Network Design				R	H	H	H					L, M	H				M	M	M		M
IT 481	Information Assurance & Security				R		L	L		L	L	L	L	M				M	M	L	M	L
Math 112	Calculus I	O		R	#																	
Math 113	Calculus II	O			#																	
Math 119	Applied Calculus	R	O	O	#																	
Math 214	Mulitvariable Calculus				#																	

1.7.3 CLOs: Course Outcomes

CLO: Course Learning Outcomes (CLOs, also called Student Learning Outcomes, or SLOs) summarize the goals and outcomes for students who successfully complete this course. These support the PLOs, but are more specific.

Course Goals and Student Learning Outcomes are as follows:

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 (scalars)
- decisions (if, else, elsif)
- looping (while, for, foreach)
- array and list variables
- subroutines

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

In teaching the major skills, I also teach the following:

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

1.8 General Topics

All syllabi are encouraged or required to address certain topics. These are generally considered to be common sense, but we find that it is useful to mention them explicitly anyway.

1.8.1 Academic Integrity

Applicable Actions

http://honorcode.byuh.edu/ details the university honor code. In the section entitled "Applicable Actions" the following are listed.

Examples of possible actions include but are not limited to the following, for instructors, programs, departments, and colleges:

Reprimanding the student orally or in writing.

Requiring work affected by the academic dishonesty to be redone.

Administering a lower or failing grade on the affected assignment, test, or course.

Removing the student from the course.

Recommending probation, suspension, or dismissal.

Depending on the specifics of the offense, any of these responses may be possible.

Cheating on exams is one form of dishonesty that I encounter. Normally this happens when students bring in notes that include answers to past exam questions. I approve the studying of past exams, and bringing in of "memories" based on study, but not the access to written notes, including notes retrieved from other exams or stored on cell phones or other devices. Any such activity, if caught, can result in failure of the entire course.

Cheating on activities also happens. Copy and paste is not allowed, but is sometimes difficult to prove. You should understand the work you submit because it helps you prepare for the exams.

Plagiarism

We learn by watching others and then doing something similar.

Plagiarism: Sometimes it is said that plagiarism is copying from one person, and research is "copying" from lots of people.

When you are having trouble with an assignment, I encourage you to look at not just one, but many examples of work done by others. Study the examples. See what you can learn from them. Do not automatically trust that they are right. They may be wrong.

http://en.wikipedia.org/wiki/Plagiarism has a wonderful article on plagiarism. Read it if you are not familiar with the term. Essentially, plagiarism is when you present the intellectual work of other people as though it were your own. This may happen by cut-and-paste from a website, or by group work on homework. In some cases, plagiarism may also create a violation of copyright law. If you borrow wording from someone else, identify the source.

Intentional plagiarism is a form of intellectual theft that violates widely recognized principles of academic integrity as well as the Honor Code. Such plagiarism may subject the student to appropriate disciplinary action administered through the university Honor Code Office, in addition to academic sanctions that may be applied by an instructor.

Inadvertent plagiarism, whereas not in violation of the Honor Code, is nevertheless a form of intellectual carelessness that is unacceptable in the academic community. Plagiarism of any kind is completely contrary to the established practices of higher education, where all members of the university are expected to acknowledge the original intellectual work of others that is included in one's own work.

CIS 101: In this course study groups are permitted and encouraged. See section 1.4.5 (page 10) for rules.

CIS 101: On exams you are required to work from personal memory, using only the resources that are normally present on your computer. This means the exams are closed book and closed notes. However, you are nearly always allowed (and encouraged!) to test your programs by actually running them on the computer where you are sitting. Students caught cheating on an exam may receive a grade of F for the semester, no matter how many points they may have earned, and they will be reported to the Honor Code office.

Faculty are responsible to establish and communicate to students their expectations of behavior with respect to academic honesty and student conduct in the course. Observations and reports of academic dishonesty shall be investigated by the instructor, who will determine and take appropriate action, and report to the Honor Code Office the final disposition of any incident of academic dishonesty by completing an Academic Dishonesty Student Violation Report. If the incident of academic dishonesty involves the violation of a public law, e.g., breaking and entering into an office or stealing an examination, the act should also be reported to University Police. If an affected student disagrees with the determination or action and is unable to resolve the matter to the mutual satisfaction of the student and the instructor, the student may have the matter reviewed through the university's grievance process.

1.8.2 Sexual Misconduct

Sexual Harassment is unwelcome speech or conduct of a sexual nature and includes unwelcome sexual advances, requests for sexual favors, and other verbal, nonverbal, or physical conduct. Conduct is unwelcome if the individual toward whom it is directed did not request or invite it and regarded the conduct as undesirable or offensive.

Brigham Young University–Hawai‘i is committed to a policy of nondiscrimination on the basis of race, color, sex (including pregnancy), religion, national origin, ancestry, age, disability, genetic information, or veteran status in admissions, employment, or in any of its educational programs or activities.

University policy and Title IX of the Education Amendments of 1972 prohibits sexual harassment and other forms of sex discrimination against any participant in an educational program or activity at BYUH, including student-to-student sexual harassment.

The following individual has been designated to handle reports of sexual harassment and other inquiries regarding BYUH compliance with Title IX:

```
Debbie Hippolite-Wright
Title IX Coordinator
Vice President, Student Development & Life
Lorenzo Snow Administration Building
55-220 Kulanui Street
Laie, Hawaii 96762
Office Phone: 808-675-4819
E-Mail: debbie.hippolite.wright@byuh.edu
Sexual Harassment Hotline: 808-780-8875
```

BYUH's Office of Honor upholds a standard which states that parties can only engage in sexual activity freely within the legal bonds of marriage between a man and a woman. Consensual sexual activity outside the bonds of marriage is against the Honor Code and may result in probation, suspension, or dismissal from the University.

1.8.3 Dress and Grooming Standards

The dress and grooming of both men and women should always be modest, neat and clean, consistent with the dignity adherent to representing The Church of Jesus Christ of Latter-day Saints and any of its institutions of higher learning. Modesty and cleanliness are important values that reflect personal dignity and integrity, through which students, staff, and faculty represent the principles and standards of the Church. Members of the BYUH community commit themselves to observe these standards, which reflect the direction given by the Board of Trustees and the Church publication, "For the Strength of Youth." The Dress and Grooming Standards are as follows:

Men. A clean and neat appearance should be maintained. Shorts must cover the knee. Hair should be clean and neat, avoiding extreme styles or colors, and trimmed above the collar leaving the ear uncovered. Sideburns should not extend below the earlobe. If worn, moustaches should be neatly trimmed and may not extend beyond or below the corners of mouth. Men are expected to be clean shaven and beards are not acceptable. (If you have an exception, notify the instructor.) Earrings and other body piercing are not acceptable. For safety, footwear must be worn in all public places.

Women. A modest, clean and neat appearance should be maintained. Clothing is inappropriate when it is sleeveless, strapless, backless, or revealing, has slits above the knee, or is form fitting. Dresses, skirts, and shorts must cover the knee. Hairstyles should be clean and neat, avoiding extremes in styles and color. Excessive ear piercing and all other body piercing are not appropriate. For safety, footwear must be worn in all public places.

1.8.4 Accommodating Special Needs

Brigham Young University–Hawai‘i is committed to providing a working and learning atmosphere, which reasonably accommodates qualified persons with disabilities. If you have a disability and need accommodations, you may wish to self-identify by contacting:

```
Services for Students with Special Needs
McKay 181
```

`Phone: 808-675-3518 or 808-675-3999`
`Email address: aunal@byuh.edu`

The Coordinator for Students with Special Needs is Leilani A‘una.

Students with disabilities who are registered with the Special Needs Services should schedule an appointment with the instructor to discuss accommodations. If the student does not initiate this meeting, it is assumed no accommodations or modifications will be necessary to meet the requirements of this course. After registering with Services for Students with Special Needs, and with permission of the student, Letters of Accommodation will be sent to instructors.

If you need assistance or if you feel you have been unfairly or unlawfully discriminated against on the basis of disability, you may seek resolution through established grievance policy and procedures. You should contact the Human Resource Services at 808-780-8875.

1.9 Syllabus Summary

Brigham Young University–Hawai‘i has adopted certain requirements relating to the information that must be provided in syllabi. This section lists those requirements and for each item either provides the information directly or gives a link to where it is provided above.

Course Information: See section 1.2.1 (page 6).

Title: Beginning Programming

Number: CIS 101

Semester/Year: Summer 2015

Credits: 3

Prerequisites: none

Location: GCB 111

Meeting Time: MWF 12:10 to 14:20

Faculty Information: See section 1.2.2 (page 6).

Name: Don Colton

Office Location: GCB 128

Office Hours: MWF 11:00-12:00.

Telephone: 808-675-3478

Email: doncolton2@gmail.com

Course Readings/Materials: See section 1.2.3 (page 6) for a list of textbooks, supplementary readings, and supplies required.

Course Description: See section 1.2.1 (page 6).

Expected Proficiencies:
See section 1.1.2 (page 5) for the proficiencies you should have before undertaking the course.

Course Goals and Student Learning Outcomes, including Alignment to Program (PLOs) and Institutional (ILOs) Learning Outcomes, and extent of coverage.

See section 1.7 (page 19) for learning outcomes, showing the content of the course and how it fits into the broader curriculum. A listing of the departmental learning outcomes is provided together with the ratings taken from department's matrix assessment document representing the degree to which the course addresses each outcome.

Instructional Methods: See section 1.5 (page 12).

Learning Management System:
`https://dcquiz.byuh.edu/` is the learning management system for my courses.

Framework for Student Learning:
See section 1.5.1 (page 13) for a discussion of the student learning framework and how I use it.

Course Calendar: See section 1.3 (page 7) for the calendar in general.

Here are some items of particular interest:

First Day of Instruction: Mon, Apr 27

Last Day to Withdraw: Thu, May 28

Last Day of Instruction: Wed, Jun 10

Final Exam: Fri, Jun 12, 12:10 to 14:20

Final Exam Location: GCB 111

Course Policies: See section 1.6 (page 15).

Attendance: See section 1.4.2 (page 9).

Tardiness: See section 1.4.2 (page 9).

Class Participation: See section 1.5.1 (page 14).

Make-Up Exams: See section 1.6.2 (page 17).

Plagiarism: See section 1.8.1 (page 23).

Academic Integrity: See section 1.8.1 (page 23).

Evaluation (Grading): See section 1.4 (page 8).

Academic Honesty: See section 1.8.1 (page 23).

Sexual Harassment and Misconduct: See section 1.8.2 (page 25).

Grievances: The university grievance policy states that the policies listed on the syllabus can act as a contract and will be considered if a student complains about the course.

Services for Students with Special Needs: See section 1.8.4 (page 26).

Chapter 2

Problem Solving

Sometimes things will not work. But all is not lost. This chapter has ideas for understanding and solving the kinds of problems we often encounter in this course.

Contents

2.1 Technical Support

The CIS department has a systems administrator to help students and faculty. GCB 119 is the best place to go for technical assistance.

Your teacher is the next best place to go for technical assistance.

2.2 cPanel Login Problems

If you are trying to use cPanel and have trouble logging in, you need technical support.

Our tech support people operate the IS2 machine where our cPanel accounts are stored.

Some of the faculty have the ability to reset passwords and make other fixes.

2.3 Programming Problems

When you run your program from the command line, it will either work or not. If it does not work, it could be because of a syntax error, or it could be because of a logic error.

Syntax errors are mistakes in the wording of how you said something. The most common syntax error is the missing semi-colon. Each statement should end with a semi-colon. If it is missing, then the program cannot be understood by the computer, and it will fail.

The computer will generally give you an error message that tells what line it got up to before it knew it had an error. Often this is the line right after the actual error. If there is an error on line 10, the computer may not notice it until it is working on line 11. Then it will report a problem on line 11, even though the actual problem is on line 10.

Logic errors (also called runtime errors or semantic errors) are mistakes in what you were asking the computer to do. Maybe you forgot to initialize a variable. Maybe you did two steps in the wrong order. Maybe you left something out.

One of the best solutions for finding and fixing logic errors is the print statement. Print out information as your program runs, things like "I got to line 12" or "the value of $i is 15". This is called debug information. It can be helpful for seeing what is going on. Compare that with what you expected. Usually that helps narrow down the mistake.

2.4 Webpage Problems

Working with an online program adds one more layer of possible confusion. Instead of going directly between your program and your user, you have a browser in the middle.

2.4.1 Webpage Does Not Load

If you are trying to load a webpage that I mention in this study guide, and the webpage does not load, it could be a DNS (domain name system) problem. In any case, the people to see are our technical support people. They run our department DNS system.

In the past this has sometimes been a problem for students living on campus, because of the special way that the CIS department is "sandboxed" to protect the rest of the university from the weird things we occasionally do. But Tech Support probably has solutions.

2.4.2 404 Page Not Found Error

404 is the error number used by the world wide web to indicate a missing webpage. If you get a **404 error** when trying to see your own webpage, it means the browser asked for the page and the server said it does not exist.

The most common cause of this problem is spelling parts of the path differently than expected.

If your webpage should be named "index.html" and you actually name it "Index.html" it will not be found. Pay attention to capitalization.

If your folder should be named "myproject" and you name it "my project" (with a space) it will not be found.

I have even seen problems when the student accidentally put a space at the end of a file name, like "index.html ".

2.4.3 500 Internal Server Error

The **500 error** means that the server found your program, and your program tried to run, but that it did not return a usable webpage.

(1) The first thing to try is this. Copy your whole web program and try running it from the command line. If there are any error messages, you can see what line is causing the first problem. Fix the first problem and then run your program again. (If there is a second problem, often it was caused by the first problem, so don't worry about it unless it is still there after fixing the first problem.)

(2) If that all looks good, then check the first few lines of program output. The first line should be "content-type: text/html". It must be spelled exactly right, no extra spaces, no blank lines in front. It must be followed by a blank line.

(3) One other common problem is file permissions. Make sure your index.cgi program has permissions of **0755** so it can be executed by the server.

2.4.4 Download Request

If you spell "content-type: text/html" incorrectly, the browser will not know that you are sending a webpage, and may ask if you would like to download the content. If you get a request like that, say "no" and then check your content-type line carefully.

2.4.5 Blank or Incomplete Webpage

Your program may run but create a webpage that is incomplete. For example, maybe it has a heading but nothing after that.

Use the "show page source" command in your browser. Sometimes that will uncover problems such as missing tag endings, like if you said `</h1` when you meant to say `</h1>`.

Chapter 3

DCQuiz: My Learning Management System

Contents

I developed my own learning management system (LMS) which we will use for this course. Other LMS examples include BlackBoard, Canvas, and Moodle. (I did not write them.) I currently do not use them.

`https://dcquiz.byuh.edu/` is the DCQuiz URL.

Since I wrote it myself, I am also responsible for any bugs that may be in its programming. If you notice any bugs, I hope you will let me know so I can get them fixed.

I can also make improvements when I think of them. I like that.

3.1 Grade Book

The most important place you will see DCQuiz is the grade book.

I use DCQuiz to manage my grade book for this class. You will be able to see the categories in which points are earned, and how many points are credited to you.

You will also be able to see how many points are credited to other students, but you will not be able to see which students they are.

This gives you the ability to see where you stand in the class, on a category-by-category basis, and in terms of overall points. Are you the top student? Are you the bottom? Are you comfortable with your standing?

3.2 Daily Update

Another place you are likely to see DCQuiz is the daily update.

Typically in class I start with a quiz called the Daily Update. It usually runs the first five minutes of class, and is followed by the opening prayer.

By having you log in and take the daily update quiz, I also get to see who is in class, in case I need a roll sheet and I did not take roll in some other way.

I also give you an opportunity to make an anonymous comment. This can be anything you want to say. It might include announcements, such as birthdays or concerts. It might include questions. It can be a simple greeting.

Comments provide a chance for each student to say something without the embarrassment of everyone else knowing who said it. You can say how unfair you think I am for something. You can ask about something you find confusing.

I introduce it something like this:

If you wish, you can type in a comment, question, announcement, or other statement at the start of class for us to consider. Or you can leave this blank.

This is a good opportunity to ask about something you find confusing.

The identity of the questioner (you) will not be disclosed to the class, and normally I will not check (although I could). My goal is for this to be anonymous.

3.3 Exams

DCQuiz was originally developed for giving tests. My problem was handwriting, actually. Students would take tests on paper and sometimes I could not read what they had written.

So I cobbled together an early version of DCQuiz to present the questions and collect the answers.

I got a couple of additional wonderful benefits, almost immediately.

First, I got the ability to grade students anonymously. All I was seeing was their answer. Not their handwriting. Not the color of their ink. Not their name at the top of the paper. It was wonderful. I could grade things without so much worry about whether every student was being treated fairly.

Second, I got the ability to share my grading results with every student in the class. Each student can see, not only the scores earned by other students, but the actual answers that other students put to each question. This gives students the ability to learn from each other.

Third, it gave students a way to verify that they were being graded fairly compared to their fellow students. If you can see your own answer, and see that everyone with higher points gave a better answer, that is a good thing. If you think your answer is better, it gives you a reason to come and see the teacher so you can argue for more points, or you can be taught the reasons for their answers getting more points.

Fourth, it gave me the convenience of grading anywhere without carrying a stack of papers. I could grade on vacation. (Wait. Doesn't that make it a not-vacation?) I could grade in class, or in my office, or at home.

Fifth, although I never did this, it theoretically has the ability for me to let other people be graders. But I never did this.

3.3.1 Taking Exams

As it currently operates, DCQuiz lets you, the student, log in and see a list of quizzes. (The grade book is actually just another quiz, but it is one where I enter grades that you earned some other way.)

Quizzes typically have starting and ending times. Before the quiz starts, there is a note telling when it will start. As the quiz gets closer, like within an hour or two, an actual count-down clock will appear telling you how long until the quiz is available.

Once you start the quiz, if it has an ending time, you will be able to see a count-down timer telling you how much time you have left.

As you take a quiz, you can see the main menu, the question menu, and the question page.

Main Menu: The main menu was already mentioned. That's where you see what quizzes are available.

Question Menu: The question menu shows you what questions are on this quiz. It lets you select a question to work on. It shows you which questions you have answered already. It shows you which answers have already been scored. It lets you say that you are done. It lets you cancel the quiz (if that is allowed).

Question Page: The question page shows a single question, and lets you type in your answer. Some questions only allow a single-line answer. When you press ENTER it takes you automatically to the next question. Other questions let you type in several lines.

Early Grading: The question page may allow an option for early grading. If you think you have given your final answer, you can submit it for early grading. If I have time, I will grade it while the test is still under way. That could give you confidence to answer related questions, knowing that you got something right.

Throwbacks: Along with early grading, I sometimes give you a second chance by doing something that I call a "throwback." That is when I look at your answer, and I think it is very close, but maybe you missed something. If so, I may unsubmit it for you. Then it will show up on your list of questions again. You can look at it and read the question again, and maybe realize what it was that you had not noticed before.

Good-Faith Testing: To get a throwback, you must have made a good-faith effort to test your program. If I see a syntax error, then I know you did not test your program, and I will not give a throwback. If I see an obvious math error, I will know that your testing was insufficient, and I will not give a throwback.

If I see an error that I think is due to overlooking some detail of the problem, then I will give a throwback. I hope you will read the question again and notice the problem.

3.3.2 Reviewing Exams

When an exam is finished, DCQuiz lets me, the author of the exam, share it with you, the student who took the exam.

You can see reviewing opportunities on the main menu.

After selecting an exam to review, you will see a question menu similar to the one that was used for taking the exam. But instead of seeing your answer, you will see all the scores that were earned, with your score highlighted. If yours is the top score, it will appear first. If it is the bottom score, it will appear last.

You can select a question to drill down and see more details. Specifically, you can see each of the answers provided by each student that wrote an answer. And you can see the score it received. And you can see any notes the grader (me) may have made while grading.

This is intended to (a) let you teach yourself by seeing examples of work by other students, and (b) let you verify that you were graded fairly. (Every once in a while, maybe a few times per semester, a student will see that I entered their grade wrong, or I overlooked something. This is your chance to get errors fixed.)

Sometimes an exam is not open for review. The teacher gets to decide. But even if the exam is closed, you can still see the question menu (with the questions blanked out), and you can see your score and everyone else's score. Questions and answers are not available, but scores are available, even long after you took the test.

Sometimes an exam is deleted or revised and reused. The teacher gets to decide. When an exam is deleted, all questions and answers and scores are also deleted. After that, there is no way to see anything about that exam.

I generally revise and reuse the daily update exams. This causes all answers and scores to be deleted, but I keep the questions and just modify them for the next class meeting.

3.4 Requesting a Regrade

If dcquiz is involved with the grading of an item, either as part of a test or as part of a grade book, it also provides you with the best way to ask for a regrade.

Dcquiz provides a Regrade button on the review screen for each problem.

When should you ask for a regrade?

Maybe I did the initial grading wrong because I was in a hurry and I just did not carefully understand what you said in your answer. If you have checked your work and you are pretty sure it is right, then press the button for Regrade and tell me why you think your answer is right. I will review your work and respond to you.

Another situation is when you wrote an online program and it was originally wrong but you have fixed it. Press the Regrade button and tell me what you have fixed. I will review your work and respond to you.

If dcquiz is not involved, so there is no Regrade button to press, you can still request a regrade by sending me an email. The required subject line is: `cis101 lastname, firstname` The body of your email should explain your thinking.

3.5 Other Features

DCQuiz has other features, such as the ability to limit where a test is taken, or to require a special code to access a test. Those features will be explained in class if they are ever needed.

Chapter 4

Activities General Information

Contents

We assume you are studying outside of class time, and that the textbook that I provide contains enough background information to avoid lots of lecturing in class.

My intention is that we will do in-class activities many times through the semester.

Chapter 7 (page 69), the Activities chapter, will be updated as new activities are assigned. Check there for activity details.

This current chapter explains the general rules that apply to each of the types of activities that will be assigned.

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 “2153 CIS 101 Activities” grade book in the column specified by the grading label.

4.1 oXX: Online General Rules

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

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

Task: Create a webpage (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.

`http://dc.is2.byuh.edu/cis101.2153/` is the student projects page URL.

4.1.1 How To Submit

Create a webpage properly linked to the student projects page. I normally grade everyone’s submissions at once, both at the 11pt (extra credit) deadline and at the 10pt (full credit) deadline.

Late Work: If you complete your work late, tell me via email so I will know to grade it. Follow the email rules in section 4.4 (page 49) in the construction of your subject line.

4.1.2 Online Requirements

Web design classes get deeply into style, also called CSS. They also cover HTML, which is the markup of the material on the webpage.

It turns out that HTML is actually very easy and can be covered relatively quickly. I have decided to provide sufficient training that you can write valid HTML, and I have decided to require it on your webpages.

Webpages consist of four main parts: content, markup, styling, and scripts. In this class we will worry only about the content and markup.

Content is what the user will see. It is typed directly into the webpage.

Markup is always specified like `<tag ...>` where tag is one of the special words that make up the HTML language. The opening and closing angle-brackets show that this is an HTML tag. Here are some tags you will use: !DOCTYPE, head, meta, title, style, body, h1, p, and img.

End tags may be required, optional, or forbidden.

Required: `<h1>` in particular requires a matching `</h1>`. An ending tag is also required for `<b>` (bold), `<i>` (italic), `<u>` (underline), `<div>` (division), `<span>` (span), and `<form>` (form).

Optional: `<head>`, `<body>`, `<p>` (paragraph), and `<li>` (list item) fall into this category. Although HTML itself does not require end tags for head and body, I do.

Forbidden: `<img>` and `<input>` do not have end tags. There is no `</input>` tag.

1. Required: **No HTML Errors:** In Firefox (the browser I use for this class), when you do a "View Page Source" request, the browser will show you the HTML of the webpage. It will also highlight important features of the webpage, and in particular it will present in bright red any errors it finds. I require that there be no errors. If I see bright red highlighting when I do a View Page Source in Firefox, it will not accept the webpage.

2. Required: **DOCTYPE:** You must have a proper DOCTYPE tag. It will be the first tag in your webpage.

   Example: `<!DOCTYPE html>`

3. Required: **head:** You must have a proper head tag. It will be the second tag in your webpage. I require an explicit head tag (and an explicit /head tag).

   Example: `<head lang=en>`

4. Required: **charset:** You must have a proper meta charset tag. It tells the browser what character set you are using. Unless you have a special need to do something else, you should use this meta charset tag. Normally this will be your third tag.

   Example: `<meta charset=utf-8>`

5. Required: **title:** You must have a proper title that includes your name and describes the task. Titles are up to about 50 characters long. Having your name early in the title makes grading easier for me.

   Example: `<title>Don's Page About Whatever</title>`

6. Recommended: **description:** Not required. Have a proper meta description tag. Descriptions are about two lines long, typically. They are used by search engines to describe your webpage. They are not the same as your title.

   If you have one, it must accurately describe your webpage.

   Example: `<meta name=description content="Don's Page About Whatever">`

7. Required: **style:** Have a proper style section, even if it is empty. Between these lines you would put any styling you will be doing. This must appear in your head section, before body.

   Example starting line: `<style>`

   Example ending line: `</style>`

   Style is often used to provide a background color or background image. It also lets you do centering and make borders. In this course you are not ever required to use style, but it is encouraged and I would be happy to tell you how to do various things.

   If you do not plan to use the style section, you can have a single line like this:

   `<style></style>`

8. Required: **/head:** Explicitly end the head section. HTML5 does not require this, but I do.

   Example ending line: `</head>`

9. Required: **body:** You must have an explicit body tag (and an explicit /body tag). The body tag will come immediately after your `</head>` tag, if you have one.

   Example: `<body>`

10. Required: **h1:** Have a proper h1 tag that identifies yourself and your project. Typically this is similar to or identical to your title. It must be ended with a `</h1>` tag.

    Example: `<h1>Don's Project About Whatever</h1>`

11. Required: **Acknowledge Professional Content:** It is best to simply avoid using professional-looking content unless it is your own. If you choose to include any professional-looking content, including for example images or text that you did not personally create, you must also include a brief statement giving credit to the source. **This is true even if you are the source.** If I think it looks professional, then it needs attribution and credit. (Take it as a compliment.) You can put your acknowledgement statement at the bottom of your page.

12. Required: **/body:** Explicitly end the body section. HTML5 does not require this, but I do.

    Example ending line: `</body>`

13. Required: **html line length:** Normally your browser does not care whether you every use `\n` on your webpage. Any kind of whitespace is pretty much the same, and it is entirely possible to make a webpage that is just one long line of html code. But I will be examining your webpages using the "View Page Source" option in FireFox, and I need to clearly and easily see your code. To this end, I require that each line of your webpage be no longer than about 100 characters so I don't have to scroll horizontally to see everything. If it is getting longer than about 100, look for a good place to split it into two or more lines. On the other hand, do not make all the lines really short either. The goal is to make it easy for me to see and understand your html code.

4.1.3 Online Template

Here is a template you can copy and modify. It includes all the required and recommended elements of your webpage.

```
<!DOCTYPE html><head lang=en><meta charset=utf-8>
<title>required title goes here</title>
<meta name=description content="optional description">
<style></style>
```

```
</head><body>
<h1>required heading goes here</h1>
put some content here
</body>
```

4.2 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) one or two other characters that specify which online task it is.

Task: Write a program. Requirements vary by task.

Follow the email rules in section 4.4 (page 49) 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 the number of points earned. Sometimes I include additional comments. You may want to save my reply until you see your grade reflected in my grade book.

4.3 gXX: GradeBot Task General Rules

GradeBot is an automatic program grader. You write the program. GradeBot runs your program and gives it sample problems to solve. Basically, GradeBot tests to see whether your program behaves properly and gets the right answers.

GradeBot itself is explained in chapter 5 (page 52).

This section explains how and what to submit for GradeBot assignments.

1. Required Comment (Customization): Each program you submit must begin with the required comment. It is the first line of the program.

It identifies the course, the assignment, and the student. It is used by GradeBot to customize your task in small ways. If you change the comment, it may change the details of your task.

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

Of course, “lastname” must be replaced with your own last name (family name), and “firstname” must be replaced with your own first name (given or personal name), as shown in my grade book.

If customizations are in effect, the background color of the GradeBot screen will turn **light green.** This will be true for most tasks.

If customizations are not in effect, the background color of the GradeBot screen will be **light yellow.** This will be rare.

2. Understand the Task: Sometimes there are specific rules to follow. These are given in the task assignment and may include things like what algorithm to use or what styling is required.

3. Test Your Program: With the required comment in place, write and test your program. Make sure you get this message:

```
Success!  No errors found.  Success report sent!
```

This means that your program runs correctly so far as we tested it. It produces the expected outputs.

4. Success report sent ...: You should also get a message at the top of your GradeBot screen that says “Success report sent to doncolton2@gmail.com for ...”, which means GradeBot has notified me that it thinks your program was successful.

This just means that GradeBot thinks your program was successful. As mentioned above, there may be specific things that would be too difficult for GradeBot to check, including your choice of algorithm or some aspects of style.

If I agree that your program is successful, I will update the grade book.

5a. Submit Your Program: If you get the the success message, but you did not get the “Success report sent” message, you can submit your program to me by sending me an email. The subject line is pretty much a copy of the required comment line. See the email rules in section 4.4 (page 49). 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.

5b. What to Send: Just send the exact program that you tested. Do a “copy and paste” from GradeBot into your email program without making any changes or additions.

Submit only your program, and not anything else. Use plain text, not rich text.

Never use “reply” to send your program. Make a fresh email. Using reply adds your email to an existing thread, and does not put it in my grading queue.

If you feel the need to make changes, do so in GradeBot. Then go back to step 3 above. Obviously, this includes creation of or changes to the required comment line.

5c. When to Send: Send it very soon after you test it. Sometimes I make changes to the program requirements or wordings. If you tested it a week ago, it may not match the current requirements.

5d. Instant Reply: If you sent your program properly, and if the Internet email system is working, you will get an instant reply telling you that your email was received. If you do not get such a reply, check your subject line.

6. Free Throwbacks (Style): When I look at your program, if there is anything obviously wrong, I will reply and let you know what to fix. Most often this is for style issues.

I will reply to your email 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.

7. Success: If your program looks okay, I will run it through GradeBot myself. If it works, I will send you a message telling you how many points you received. Save that message until you see your score show up in my grade book.

8. Penalty Throwbacks (Does Not Work): If I run your program through GradeBot and it fails, I will tell you so. I may also impose a one-point penalty for each time you submit a program that does not work. I will assume that you did not test it in GradeBot. If it works for you, it should work for me.

4.4 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.

4.4.1 To: Line

Please send email to `<doncolton2@gmail.com>`. That is my preferred email address.

If you cannot use that, you are welcome to 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.

4.4.2 Subject Line

The subject line of the email must be as follows:

```
cis101 label 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 `cis101` part must be written exactly as shown. It is a magic word that lets the email system identify your message as something that belongs to this class. It keeps it out of my spam folder. Warning: Do not add a space in the middle of it, for example, or your email will not go to the right place.

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

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

The `firstname` part is to be 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
```

4.4.3 Instant Response

When you have properly submitted an email message to me (correct email address, and correct magic word "`cis101`" in the subject line), you will receive an automatic reply almost immediately. This proves that I received your message. If you do not receive such a reply, it may mean that I did not receive your message.

Here is a typical reply:

> This is an instant reply to let you know that I received your email relating to CIS 101 (Beginning Programming), and it has been placed into my pending file for review and/or grading as soon as I can get to it. You can expect another reply from me when I have looked at your email. Thanks! Bro Colton

If you want your email treated as a formal communication to me, look for the instant response.

4.4.4 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, `#` is used to start a comment line. So, following the previous example from above, I would have this comment line as the first line of my program.

```
# cis101 p1 Colton, Don
```

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.

Also, please make it easy for me to tell where your program begins and ends. If you have anything else in your email, make sure it is clearly separated from your program.

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.

Chapter 5

GradeBot

GradeBot is my automated program grader.

Contents

I normally grade programming activities based on the following three criteria.

Behavior: How does the program respond to various situations? This is always important.

Style: How clearly is the program written? My standards vary from course to course and from assignment to assignment.

Algorithm: What algorithms are used? Were they efficient? My standards vary from course to course and from assignment to assignment.

When I grade on algorithm, I rely on visual inspection of the program source code.

When I grade on style, I rely on a combination of GradeBot and visual inspection of the program source code.

When I grade on behavior, I usually rely on GradeBot. GradeBot verifies that your program seems to be running correctly by giving your program test cases to solve, and then seeing whether your program returns the correct answer each time.

(I have used GradeBot for many years. This particular version of GradeBot is sometimes called GradeBot Lite because it is descended from what used to be a huge system that was also called GradeBot.)

5.1 Where Can I Find GradeBot?

`http://gradebot.tk/` is the web interface for GradeBot.

`http://gbot.is2.byuh.edu/` is an alternate URL that you can use in case the short URL stops working or does not work for you.

If you want to explore, press the `[List All Labs]` button. Then pick a lab from the list and press its button.

GradeBot will give you a "SAMPLE EXECUTION" to show the behavior it is expecting from your program.

5.2 Source Code File

To keep things simple, GradeBot requires you to submit a single file of source code. You are not permitted to write modules as separate files, and compile them separately and then link the results. Such abilities are present in Integrated Development Environments, but not in GradeBot. Everything must fit into a single file. In some languages, this can be an uncomfortable or even unacceptable restriction. Sorry.

GradeBot also allows you to type your program directly into the web interface. For short programs, that is probably what happens most often. GradeBot also allows you to upload your file by selecting it on your local computer.

5.3 Choice of Language

GradeBot is able to accept programs in any of several different languages. The list includes C, C++, Java, Perl, Python, Ruby, and Tcl. Other languages may be added. Feel free to make a request.

You must tell GradeBot what language you are using. Currently GradeBot defaults to Perl. If you are using another language, just click on the appropriate radio button.

When grading your program, GradeBot will compile or interpret your program and then check its behavior. If your program has syntax errors, GradeBot will let you know. If there are no syntax errors, GradeBot will run your program. If you request Indent Checking, GradeBot will also look at your indenting to see whether it seems consistent. I often require consistent indenting.

Java: This popular language is perhaps the most disadvantaged with GradeBot for several reasons: (a) Java likes to have separate files; (b) Java takes a long time (half a second) to start, so 20 tests will take 10 seconds if you are lucky. The bottom line is that Java can be used, but it is a bit harder to make things work.

5.4 Customizations

GradeBot can customize its requirements for each student. This is done based on the first line of the program, which may be a required comment line.

If that first line includes a particular string, the whole line is used to seed a random number generator that creates the differing requirements for each student.

For example, the instructor may require each program to include a comment line that identifies the task and the student, something like this:

```
# cis101 g05 Colton, Don
```

GradeBot may be looking for the string `cis101 g05` as its trigger for customizing.

Because these customizations are intended to be different for every possible first line, the result is that every student may have to create a slightly different program.

It is an honor code violation to submit a comment line that makes it look like you are somebody else, especially somebody else in the class. Other than that, it is okay to mess around with the comment line.

Changing the first line can change the customizations, so after you start working on a program, you should be careful about changing the first line. The other lines of the program do not affect the customizations in any way.

An example of a customization is the string the student must use to prompt for the first input. GradeBot might select from among the following options.

```
Please enter a number.
Kindly enter a number.
Please type in your number.
Type in a number please.
```

The purpose of this customization feature is to cut down on simple copying that might go on among students. Without customization, the same program could be submitted by every student. With customization every student would need to adjust the behavior of their program a bit to match the expectations of GradeBot.

5.5 GradeBot Log File

GradeBot keeps a log of programs submitted and their results. Currently it keeps the first line of the program (the comment line) together with the name of the lab being attempted and the results of testing (success or not). This information is available to me and can be used to give students credit for completing work even if they have not emailed it to me. For this reason, it is an honor code violation to submit a comment line that has someone else's name on it.

For programs that GradeBot thinks are successful, it also sends me an email that includes the full program. This information can be used to give students credit for completing work even if they have not emailed it to me.

Together these features reduce the need to email programs to me, but there may still be cases where emailing is required.

5.6 Standard In, Standard Out

Rigorous testing of computer programs is actually very difficult. To make it possible, I have made some decisions to keep things as simple as possible.

Tasks are generally limited to programs that read input from STDIN and write output to STDOUT. Beyond that, it can provide input through command line arguments, and it can inspect the return value from the programs it is testing.

That means that GradeBot does not get involved with mouse-based input, or network-based input or output, or graphical outputs.

Generally GradeBot does not get involved with reading and writing files, or accessing databases.

GradeBot also limits the amount of time each program is allowed to run.

5.7 The Grading Script

GradeBot has a version of each program you are asked to write. It generates sample inputs (somewhat randomly), runs its own version of the program, and collects the outputs. That is used to make a script: say this to the program, wait for the program to say this, repeat.

Your program is tested by seeing whether it can follow the script. Your program must behave exactly the same as GradeBot's program did.

This puts some serious constraints 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. Usually it is a minor typographical problem.

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. That means your program's observable behavior is correct.

5.8 Understanding GradeBot's Requirements

GradeBot is very picky. That is because it is really hard to tell when something is almost right, or close enough. GradeBot's only real choice is to require absolute perfection. That means spelling and spacing must be exact.

GradeBot shows you exactly what it wants, and exactly what you provided. Sometimes it can tell you what is wrong, but often you have to compare things and figure it out yourself. Just do a careful side-by-side comparison and adjust your spelling and spacing to make GradeBot happy.

(GradeBot may actually do things wrong in some cases. Maybe GradeBot spells something wrong or uses the wrong grammar. If you run into a case like that, the simple solution is to play along and do it the way GradeBot wants. You can also let me know where GradeBot is wrong and I may be able to fix it. I may even give extra credit for the mistakes you report.)

Standard Input

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

Standard Output

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

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 of output ends with a newline character unless specified otherwise.

"eof" stands for end of file and means that your program must terminate cleanly.

Command Line Inputs

GradeBot will start your program by saying this:

```
GradeBot started your program with this command line:
```

It will then present the command line that was used to start your program. Often there is no special command line input, and the command line simply starts your program. Sometimes there are command line arguments. Here is an example:

```
"./gcd 35 28"
```

In this example, `./gcd` is the name of your program that is being run. `35` is the first command line argument. `28` is the second command line argument.

In most languages, command line arguments are available as an array named “argv” or “args” or something similar.

Return Codes

When your program terminates, it must send back a return code of zero unless something else was specified in the requirements. Many languages do this automatically.

For C programs, remember to start your program with “int main” and end it with “return 0;”

For other languages, do something similar if necessary.

Chapter 6

Programming Style

As your programs become more complex, style becomes important.

Contents

You should use good style. I consider programming style to be very important. I have different rules for CIS 101 and for the other classes I teach. This chapter explains those rules.

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 May, Should, and Must

In the requirements mentioned here and elsewhere, I use the words "may," "should," and "must" (and sometimes "should not" or "must not") to describe the importance of requirements.

I try to be careful in my use of these words so you can trust them.

When I say "may" or "can" it means that you are permitted but not encouraged nor required to do that particular thing.

When I say "should" it means that the thing is optional but strongly encouraged.

When I say "must" it means that the thing is fully required, and that without compliance, you will not receive credit.

6.2 In Every Class

Style is really all about making your program easy to read and update. That is what I truly care about. I have often seen programs that were buggy because they did not follow good style rules, and as a result the programmer had become confused.

Indenting: Indenting **should** correctly reveal the internal structure of your program. It **must** be consistent (a similar amount of indenting throughout your program) and reasonable (one or more spaces or tabs).

Naming: Variable names **must** helpfully describe the contents they hold. Subroutine names **must** helpfully describe what they do.

Comments: You **should** use comments to explain your code if things are not immediately obvious. Use comments to explain what you are trying to accomplish with the key paragraphs of your program. Use comments to explain anything that might be hard to understand in the future.

Your code **should** be readable to a programmer even if they are unfamiliar with your particular programming language. Most programming languages are similar enough that any programmer can read them. (Writing, on the other hand, can often require memorization of syntax rules.)

If I complain about your style, what it probably means is I had trouble understanding your program. It may run fine in the computer, but it was difficult for me.

6.3 In CIS 101

In CIS 101 I have additional rules to help students learn good habits.

6.4 Spacing

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

Example: `(15+$x)` is bad because it does not have enough spaces.

Example: `( 15 + $x )` is good because the spacing is perfect.

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

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 **should** omit the space before a semi-colon.

Example: `$x = 23 + $y;` is okay.

Example: `$x = 2 3 + $y;` is bad.

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

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

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

Exception: You **should not** put a space between the dollar sign (or at sign) and the rest of the variable name.

Example: `$x` is okay.

Example: `$ x` is bad.

6.5 One Statement per Line

Each statement **should** be on its own line.

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

Exception: A relevant comment **may** be placed after a semi-colon or opening curly brace.

Exception: Short statements are generally okay after the opening curly brace. I will accept things like these:

```
if ( $x == 5 ) { last }
```

```
if ( $x == 5 ) { $y++ }
```

```
else { print "goodbye" }
```

Exception: The `for` loop uses two semi-colons to organize its control structure (initialize; condition; step). You **must not** start a new line after those semi-colons unless it seriously improves the clarity of the code.

6.6 For Loops

The “for” loop uses semi-colons, so I am talking about it here, right after we talked about semi-colons.

There are a few special style rules that apply to for loops. The for loop has a specific control structure that is a bit different from other parts of the program. All the controls are specified at the top of the loop. In contrast, with a while loop, the init happens before the loop and the step happens at the end of the body.

The layout of a for loop is like this:

```
for ( init; cond; step ) { body }
```

The init (initialization) is an actual statement. You **may** have more than one statement, and separate them by commas.

The cond (condition) is not a statement, but is a condition like we would find with an if statement or a while loop.

The step is an actual statement. You **may** have more than one statement, and separate them by commas.

Taken together, init, cond, and step are the “controls” of the for loop, and style-wise they **must** be placed at the start of the loop to clearly indicate how the loop starts, stops, and moves forward. Normally they are all given on the same line, along with the word “for” itself. They **must not** be broken up on separate lines unless there is a special reason.

Because the controls should be kept together, it is bad style to do the init activity before the for loop, and bad style to do the condition inside the loop unless it is necessary, and bad style to do the step activity inside the body. Any one of these would violate the style requirements of the for loop.

Order of the condition: Most for loops are controlling a single variable, something like this.

Good: `for ( $x = 1; $x < 10; $x++ ) { body }`

We could also write it like this, but this is much harder to read because we expect the variable to be the first thing in each slot.

Bad: `for ( $x = 1; 10 > $x; $x++ ) { body }`

6.7 Indenting

The “indent” is the number of blanks at the start of each line.

Summary: Your program **must** start out with no indenting. Each time you have an opening curly brace or opening parenthesis, you increase your indent. Each time you have a closing curly brace or closing parenthesis, you decrease your indent.

I use and recommend two spaces for the indent. You **must** use one or more spaces, or a tab, for your indent. You **must** be consistent. Decide on your indent and use it everywhere an indent is needed.

Crazy indenting makes programs substantially harder to read, so I am picky about this and GradeBot enforces it.

The main program **must 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, the indent **must** be increased by one step.

The opening { **should** be at the end of a line, and not on a line by itself. Normally it will be part of an if statement or a loop.

The closing } **should** be at the end of the last line in the block, and not on a line by itself. If there are two }s they **should** be on the same line.

The following is a good example.

```
if ( $x < 5 ) {
  print "$x is less than 5" }
```

The following is a **bad** example because the closing brace is on the next line.

```
if ( $x < 5 ) {
  print "$x is less than 5"
  }
```

The following is a **bad** example because the opening brace is on the next line.

```
if ( $x < 5 )
  {
  print "$x is less than 5" }
```

6.8 Use the Stated Values Exactly

Often a problem will specify certain numbers or strings that define how the program should run. If possible, you **must** use those exact same values in writing your program. If not, you **must** include a nearby comment that states 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 you **must** 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`

6.9 Put Quotation Marks On Literal Strings

You **should** always put quote marks on literal strings. A literal string is a sequence of letters, possibly including interpolated variables.

It is wrong (but possible) to use strings without quotation marks.

Example: `$x = random; # $x will be "random"`

Example: `$x = currenttime; # $x will be "currenttime"`

Sometimes it fails to give the expected answer.

Example: `$x = rand; # $x will not be "rand"`

Example: `$x = localtime; # $x will not be "localtime"`

That is because unquoted strings, also known as barewords, can be interpreted to be function names or subroutine names. If nothing matches, then they are normally understood to be strings, but this cannot be relied upon.

Because the meanings cannot be relied upon, I strongly frown on this casual use of unquoted strings. I expect them to be properly quoted. Partly this is a style issue. Partly it goes deeper, as it avoids ambiguity.

6.10 Avoid Useless Statements

Every statement **must** make a meaningful contribution to the success of the program. Meaningless statements add confusion to the program, as future programmers are wondering why those statements exist.

One example is the useless chomp.

```
chomp ( $x = $y + 5 );
```

In this example, `$x` will not have a trailing newline because it is the result of a mathematical operation. The chomp is to remove trailing newlines, but one could never exist in this setting. Therefore the chomp is useless.

Another example is the useless initialization in a for loop:

```
for ( $x = $x; $x < 100; $x++ ) {
```

In this case, `$x` was apparently initialized before the loop (which is a style error itself), and the `$x = $x` is useless. You are copying x into x, which changes nothing but makes the program run slower (ever so slightly). In this case, the loop should be written like this:

```
for ( ; $x < 100; $x++ ) {
```

If you include a statement that is not necessary for the correct operation of the program, you **should** also include a comment that explains why that statement is there.

6.11 Use Helpful Names

Look in the textbook index for “variable naming.”

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 **must** be helpful. They **must** bear some obvious relationship to the thing they represent.

Long descriptive names **may** 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.

Ambiguous names are not good. Avoid ambiguity.

Example: If you are converting temperature from Fahrenheit to Celsius, and you call one of your variables `$temp` or `$temperature` it is bad because it is not clear whether the information in that variable is the temperature in Celsius or in Fahrenheit. Variable names **should** give a clear indication of the type of information held inside.

Example: If you are asking for the quantity of engines and the cost of engines, then `$engine` is **not** a good variable name because one cannot tell whether you are talking about the quantity of engines or the cost per engine or maybe the total cost of that quantity of engines. Be clear.

6.12 Parentheses: Math vs Array

You **should not** use outer parentheses when calculating are assigning values.

You **should not** say `$x = ( something );` because the parentheses make it ambiguous. It has two possible meanings. Perl usually handles it okay, but I frown on it.

First Meaning: Mathematical Grouping

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

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

In the next example, we have added some outer parentheses. These outer parentheses do not help anything. They do not clarify anything. They do not change the meaning of the equation.

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

Second Meaning: Defining an Array or List

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? You **should not** put "outer" parentheses around a whole expression or statement.

6.13 Limited Use of Blank Lines

Blank lines **should not** be used in the program unless they improve the clarity of the code.

Sometimes it is helpful to add a blank line here or there to divide longer sections of code into natural paragraphs. It can show that some lines fit together as a natural block of related programming.

6.14 Penalty Limits

Style is very important for promoting clarity in programs, and should be a goal of every programmer.

Most daily activity assignments are worth 10 points if completed on time. They are worth 7 points if completed late but before the end of semester deadline.

For activity programs that would have earned 10 or more points except for style errors, rather than flatly rejecting the program I may award 7 points. I will also allow the student to resubmit the program with style errors corrected for a possibly higher score. For programs that would have earned 7 or fewer points, we will award those points despite the style errors.

For exam programs where style is required, style errors will continue to result in a throwback with comment visible after the exam. This will also depend on how serious the error is, in comparison to the rest of the problem.

Chapter 7

Activities Assigned

Activities give you a way to develop and test your programming skills.

This chapter lists the activities that have been assigned. As new activities are assigned, they will be added to this chapter.

Reminder: You **must not** give a copy of your program to another student. You are cheating them out of the learning experience that they should be having. Please limit your assistance to these things:

(a) You **may** let them look at your program, but not make notes. If they can memorize it, I guess that is okay, but if they are writing things down, or if they take a picture of it, they have committed an honor code violation. (It is okay to take pictures of things I write on the white board and to share them.)

(b) You **may** look at their program and give them advice. Tell them where you think their errors are. Do not tell them what to type. Say things like, "you might need to change that," or "are those supposed to be the same?" or "you might need something at the end of that line." Make them do some thinking.

Contents

7.1 o1H: First Webpage

- Discussed: Mon, Apr 27
- 11pt Due Date: Mon, Apr 27, 23:59.
- 10pt Due Date: Wed, Apr 29, 23:59.
- 7pt Due Date: Wed, Jun 10, 23:59.
- Grading Label: **o1H**

Online: 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 4.1 (page 42) apply. To request a regrade, see 3.4 (page 40).

We normally do this in class the first day. If you miss class that day, please work with one of the tutors to complete the assignment.

Log into cPanel. Go into File Manager.

Go into your `public_html` directory (folder).

You **must** create a directory (folder) with `cis101.2153` as its name. The cis101 part stands for this class. The 2153 part stands for this semester.

In that directory you **must** create a file with `index.html` as its name. Be careful that you do not create `INDEX.HTML` or anything else that is not `index.html` exactly.

Construct a webpage. You **must** have at least this content or more: head, title, body, h1, image.

You **must not** have any of these problems: any HTML syntax error, copied content that is not properly updated.

You **may** use the following lines as an example. You **may** use cut-and-paste if you want, but you **should** type them in so you start thinking about what they mean. If you have more HTML skills than this, feel free to do a more impressive job.

```
<!DOCTYPE html><head lang=en><meta charset=utf-8>
<title>Don’s CIS 101 Homepage</title>
<style></style>
</head><body>
<h1>This is Don’s CIS 101 Homepage</h1>
<img src=mypic.jpg alt="my picture" width=500>
</body>
```

You **must** change the name "Don's" to match your own name, of course.

You **must** upload a picture "of" yourself. It can simply be something that represents you, or it can be an actual picture of yourself. You **may** name it whatever you want, so long as the webpage works correctly.

By upload, I mean that you **must** actually upload a picture and not simply link to a picture that already exists on another website.

Verify that you can reach your page through clicking on your o1H link on the CIS 101 Student Projects webpage, which is:

`http://dc.is2.byuh.edu/cis101.2153/`

7.2 g21: Hi Fred

- Discussed: Mon, Apr 27
- 11pt Due Date: Mon, Apr 27, 23:59.
- 10pt Due Date: Wed, Apr 29, 23:59.
- 7pt Due Date: Wed, Jun 10, 23:59.
- Grading Label: **g21**

GradeBot: This is a GradeBot task. If your program runs correctly and has proper indenting, GradeBot will send it to me for you. The general rules and explanations in section 4.3 (page 46) apply, including the required comment line.

`# cis101 g21 lastname, firstname` is the required comment line. There may be customizations that are different for each student based on this comment line. Your wordings may differ from study guide examples.

GradeBot is at `http://gradebot.tk/` and `http://gbot.dc.is2.byuh.edu/`.

There is a chapter in the textbook, probably about chapter 4, probably about page 35, that talks about accepting inputs.

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 is your name?"
 in>  "Fred"
  2:  "Hey, Fred!"
 eof  (end of output)
```

7.3 g33: Bank Balance

- Discussed: Wed, Apr 29
- 11pt Due Date: Wed, Apr 29, 23:59.
- 10pt Due Date: Fri, May 01, 23:59.
- 7pt Due Date: Wed, Jun 10, 23:59.
- Grading Label: **g33**

GradeBot: This is a GradeBot task. If your program runs correctly and has proper indenting, GradeBot will send it to me for you. The general rules and explanations in section 4.3 (page 46) apply, including the required comment line.

`# cis101 g33 lastname, firstname` is the required comment line. There may be customizations that are different for each student based on this comment line. Your wordings may differ from study guide examples.

GradeBot is at `http://gradebot.tk/` and `http://gbot.dc.is2.byuh.edu/`.

You will be asked for a beginning balance, deposit total, and withdrawal total. You must calculate and display the ending balance.

7.4 o1R: Random Number

- Discussed: Wed, Apr 29
- 11pt Due Date: Wed, Apr 29, 23:59.
- 10pt Due Date: Fri, May 01, 23:59.
- 7pt Due Date: Wed, Jun 10, 23:59.
- Grading Label: **o1R**

We will be doing an online program. We will do it in two steps. First we will make a command-line program and get it working. Then we will modify it to run on the web.

Online: 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 4.1 (page 42) apply. To request a regrade, see 3.4 (page 40).

There is a chapter in the textbook, probably about chapter 9, probably about page 68, that talks about random numbers.

Summary: Make a web program that displays a random number each time you load it.

Warning: I am going to give you some of the information that you need. You need to figure out what order things must happen. You are welcome to confer with your neighbors, but do not simply trust them and copy their work.

7.4.1 Step 1: Simple Command Line Version

You do not turn in this step. You turn in step 3. But step 1 is important for testing your program.

Write the following program (or one like it) and test it.

```
$random = rand(30);
print "Your number is $random.";
```

That will create and display a random number between zero and 30.

Run your program at the command line. Each time you run it, you should see a new random number.

7.4.2 Step 2: Command Line Version

You do not turn in this step. You turn in step 3. But step 2 is important for testing your program.

When your program is executed, it must print out the HTML code for a webpage.

You can use the following webpage as an example. Cut-and-paste might work but it is better to type it in so you start thinking about what it means. If you have more HTML skills than this, feel free to do a more impressive job.

Make sure you have at least this content or more: doctype, head, title, body, h1, random number.

The webpage must start with `<!DOCTYPE html>`

The title must have your firstname first.

The h1 must have your firstname first.

The webpage must end with `</body>` and have no obvious errors.

```
<!DOCTYPE html><head lang=en><meta charset=utf-8>
<title>Don's Random</title>
<style></style>
</head><body>
<h1>Don's Random Number Generator</h1>
$random
</body>
```

Remember: Unless your name is Don, it should not say Don in it.

Remember: To print a quote mark, use a backslash in front of it.

You have to pick the right time for this number to be generated.

Run your program at the command line. Each time you run it, you should see the HTML version of a web page displayed.

7.4.3 Step 3: OnLine Version

File name: Log into cPanel. Go to your cis101.2153 folder. Create a sub-folder with `random` as its name. In it create a file with `index.cgi` as its name.

Permissions: This `index.cgi` file will be a program. Because it is a program, you must set its permissions to **0755**. Normal webpages have their permissions set to **0644**.

The **0755** permission tells the is2 machine that this will be a program, and the perl line tells it that it is a perl program (as opposed to php or ruby or something else).

Add these two lines to the front of your program.

```
#! /usr/bin/perl --
print "content-type: text/html\n\n";
```

The rest of your program can be exactly like the command line version. Copy and paste and save it.

Run your program by viewing your webpage. Each time you do a reload on your webpage, it will cause your program to run again, and a new number will appear.

Verify that you can reach your page (and run your program) through clicking on your o1R link on the CIS 101 Student Projects webpage, which is:

`http://dc.is2.byuh.edu/cis101.2153/`

If you get a 404 File Not Found error, see section 2.4.2 (page 33) for guidance.

If you get a 500 Internal Server Error, see section 2.4.3 (page 33) for guidance.

7.5 g35: Age

- Discussed: Fri, May 01
- 11pt Due Date: Fri, May 01, 23:59.
- 10pt Due Date: Mon, May 04, 23:59.
- 7pt Due Date: Wed, Jun 10, 23:59.
- Grading Label: **g35**

GradeBot: This is a GradeBot task. If your program runs correctly and has proper indenting, GradeBot will send it to me for you. The general rules and explanations in section 4.3 (page 46) apply, including the required comment line.

`# cis101 g35 lastname, firstname` is the required comment line. There may be customizations that are different for each student based on this comment line. Your wordings may differ from study guide examples.

GradeBot is at `http://gradebot.tk/` and `http://gbot.dc.is2.byuh.edu/`.

You will be asked for an age in years. You will convert it into months, days, hours, minutes, and seconds.

7.6 o1D: Pair O Dice

- Discussed: Fri, May 01
- 11pt Due Date: Fri, May 01, 23:59.
- 10pt Due Date: Mon, May 04, 23:59.
- 7pt Due Date: Wed, Jun 10, 23:59.
- Grading Label: **o1D**

Online: 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 4.1 (page 42) apply. To request a regrade, see 3.4 (page 40).

There is a chapter in the textbook, probably about chapter 9, probably about page 68, that talks about random numbers.

Summary: Make a web program that rolls two multi-sided dice randomly each time you load it.

Make sure you have at least this content or more: doctype, head, title, body, h1, two dice, and no html errors (bright red in FireFox).

You might have to look at previous assignments, specifically the o1R assignment, for some how-to information that applies to this task.

In the proper location, create a sub-folder with `dice` as its name. In it create a file with `index.cgi` as its name.

When your program is executed, it will print out a webpage. You can have it print a webpage that looks something like this:

```
content-type: text/html

<!DOCTYPE html><head lang=en><meta charset=utf-8>
<title>Don’s Dice Roller</title>
<meta name=description content="roll two dice">
<style>
body { background-color: #35ffff; text-align: center; }
</style>
</head><body>
<h1>Don’s Dice Roller</h1>
<p>We will roll two dice.
<p>We rolled a 1 and a 6.
<p>
```

```
<img src=1.jpg alt=1 title='we rolled 1'>
<img src=6.jpg alt=6 title='we rolled 6'>
</body>
```

Instead of printing an actual 1 and 6, etc., as shown, use variables, maybe $d1 and $d2 for dice 1 and dice 2, etc.

```
$d1 = 1 + int ( rand(6) );
```

That will create a random integer between 1 and 6.

The `rand(6)` part will generate a number between zero and six, but it will never actually be six. One sixth of the time it will be between 3.000 and 3.999 (more or less).

The `int` part converts a number like 3.14159 into an **int**eger like 3 by cutting off the fractional part of the number. It is like chomp, but for the fractional parts of numbers.

After int happens, one sixth of the time the result will be exactly 3.

Run your program by reloading your webpage. Each time you do a reload on your webpage, it will cause your program to run again, and a new set of dice will appear.

You will also need images of dice. You are welcome to copy my dice from my webpage and upload them to your own webpage.

Feeling creative? You can use another kind of dice (like a d8 or a d20). You can use other images, maybe star, bell, cherry, rainbow, etc., like a slot machine. But you must have at least six image choices, and exactly two displayed at a time.

7.7 g41: Gift

- Discussed: Mon, May 04
- 11pt Due Date: Mon, May 04, 23:59.
- 10pt Due Date: Fri, May 08, 23:59.
- 7pt Due Date: Wed, Jun 10, 23:59.
- Grading Label: **g41**

GradeBot: This is a GradeBot task. If your program runs correctly and has proper indenting, GradeBot will send it to me for you. The general rules and explanations in section 4.3 (page 46) apply, including the required comment line.

`# cis101 g41 lastname, firstname` is the required comment line. There may be customizations that are different for each student based on this comment line. Your wordings may differ from study guide examples.

GradeBot is at `http://gradebot.tk/` and `http://gbot.dc.is2.byuh.edu/`.

Style: Approved style is required on this task. The rules in chapter 6 (page 59) apply. Be careful with your spacing, indenting, and choice of variable names.

There is a chapter in the textbook, probably about chapter 20, probably about page 112, that talks about numeric comparisons.

Task: See whether you can afford a certain gift or not.

Sample Execution:

```
  1:   "How much money do you have? " (no \n)
 in>    ..........................."1.00"
  2:   "How much does the gift cost? " (no \n)
 in>    ............................"2.00"
(next line depends on the numbers)
  3:   "Sorry. You cannot afford it."
  3:   "Perfect. You can afford it."
 eof  (end of output)
```

7.8 c2M: Mad Lib

- Discussed: Mon, May 04
- 11pt Due Date: Mon, May 04, 23:59.
- 10pt Due Date: Fri, May 08, 23:59.
- 7pt Due Date: Wed, Jun 10, 23:59.
- Grading Label: **c2M**

Command Line: This is a command-line task. The general rules in section 4.2 (page 46) apply, including email subject line and program comment line.

`cis101 c2M lastname, firstname` is the required subject line.

`# cis101 c2M lastname, firstname` is the required comment line.

There is a chapter in the textbook, probably about chapter 4, probably about page 35, that talks about accepting inputs.

A "Mad Lib" is a fill-in-the-blank story. You start by asking for a list of words and then you insert them into the blanks of a story. The result is sometimes very funny.

Task: Write a mad lib. Prompt for 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.

Requirement: You must invent the story yourself. You cannot get it from any other source.

Requirement: There must be at least three inputs and at most five.

Requirement: Each input must be used at least twice in the story.

Request: The story should be about two paragraphs long.

Request: Please make the story creative and interesting.

Common Problem:

You may want to put 's (or some other letters) after one of your inputs. This is tricky because those things can be seen as part of the variable name even though they are not. To solve this problem, put the variable name in curly braces, like this:

```
${x}'s
```

Or you can escape the apostrophe with a back-slash, like this:

```
$x\'s
```

7.9 g42: Birthday

- Discussed: Fri, May 08
- 11pt Due Date: Fri, May 08, 23:59.
- 10pt Due Date: Mon, May 11, 23:59.
- 7pt Due Date: Wed, Jun 10, 23:59.
- Grading Label: **g42**

GradeBot: This is a GradeBot task. If your program runs correctly and has proper indenting, GradeBot will send it to me for you. The general rules and explanations in section 4.3 (page 46) apply, including the required comment line.

`# cis101 g42 lastname, firstname` is the required comment line. There may be customizations that are different for each student based on this comment line. Your wordings may differ from study guide examples.

GradeBot is at `http://gradebot.tk/` and `http://gbot.dc.is2.byuh.edu/`.

Style: Approved style is required on this task. The rules in chapter 6 (page 59) apply. Be careful with your spacing, indenting, and choice of variable names.

Unit 4 of the textbook talks about numeric decisions. This program will use that knowledge.

Task: Tell how many years old a person is. This may involve several if statements.

Sample Execution:

```
GradeBot engaged your program in this dialog:
note  GBot "# debug output lines are permitted"
note  GBot "# For this program, assume today is Aug 15 2014."
  1:  "Please enter your name: " (no \n)
 in>   ........................"Michelle"
  2:  "What month (number) were you born, Michelle? " (no \n)
 in>   ............................................"10"
  3:  "What day were you born, Michelle? " (no \n)
 in>   ................................."25"
  4:  "What year were you born, Michelle? " (no \n)
 in>   .................................."1984"
  5:  "Ah. You were born on 1984-10-25."
  6:  ""
  7:  "Michelle, you are 29 years old."
 eof  (end of output)
```

7.10 o2M: Mad Lib

- Discussed: Fri, May 08
- 11pt Due Date: Fri, May 08, 23:59.
- 10pt Due Date: Mon, May 11, 23:59.
- 7pt Due Date: Wed, Jun 10, 23:59.
- Grading Label: **o2M**

Online: 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 4.1 (page 42) apply. To request a regrade, see 3.4 (page 40).

There is a chapter in the textbook, probably about chapter 10, probably about page 72, that talks about accepting online input.

Task: Make an online Mad Lib story generator.

This is similar to the c2M Command-Line Mad Lib project we did earlier, but the inputs are online.

As for all online programs, your program must be properly linked to the student projects page, and the displayed webpage must show your name.

Recommended (not Required):

Use the same story you used for the command-line MadLib task.

Put your program in this order: (1) /usr/bin/perl line, (2) olin subroutine, (3) all of your olin subroutine calls, (4) any decisions or calculations, (5) exactly one very big print statement that prints everything needed, from content-type through the end.

Requirements:

(a) Story: Use a web program to tell a story that involves user-provided inputs. Properly handle multi-word inputs.

(b) 3 to 5 Inputs: Your program must display and accept input from at least three and at most five input fields. Each must have a helpful and suitable description. Each value must be used at least twice in your story.

(c) Autofocus: Your first input field must have autofocus.

(d) Submit: Your program must have a submit button. When the submit button is pressed, the screen should be updated.

(e) Defaults: When your program runs, if user inputs are provided they must be used. Otherwise, suitable default values must be used. Each input must have a reasonable (non-blank) default value.

(f) Sticky: When the screen is updated, the values that were used shown must be shown in the input fields. (They must be "sticky.") Make sure it works for multi-word inputs.

(g) Bold: Each time an input value is used in your story, you must make it obvious. If they simply blend in it makes it hard for me to find them. The recommended way is to surround its use with bold markup. Instead of saying `$x` say `<b>$x</b>`.

(h) olin: We provide the olin subroutine to help you retrieve your inputs.

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, but verify that it copied correctly, especially the quote marks. Look in the textbook index for "olin" to find an explanation of this subroutine.

Since you will use this on several assignments, it might be worth your time to clean up the indenting errors that often come with cut and paste.

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. I recommend putting it near the top. Then include one or more calls to olin as shown here.

You normally call olin with two parameters, like this:

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

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

Notice that `chomp` is not used with olin. It is not needed. chomp is mostly used with STDIN with keyboard inputs.

Although it is not recommended, 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. Because it is harder to work with, this is probably not a good approach.

For debugging purposes, 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. When you are debugging it can be very informative to see what that input string really looks like. For example:

```
print olin;
```

7.11 g51: Phone Book

- Discussed: Mon, May 11
- 11pt Due Date: Mon, May 11, 23:59.
- 10pt Due Date: Wed, May 13, 23:59.
- 7pt Due Date: Wed, Jun 10, 23:59.
- Grading Label: **g51**

Task: Tell what page of the phone book has the name you seek.

GradeBot: This is a GradeBot task. If your program runs correctly and has proper indenting, GradeBot will send it to me for you. The general rules and explanations in section 4.3 (page 46) apply, including the required comment line.

`# cis101 g51 lastname, firstname` is the required comment line. There may be customizations that are different for each student based on this comment line. Your wordings may differ from study guide examples.

GradeBot is at `http://gradebot.tk/` and `http://gbot.dc.is2.byuh.edu/`.

Style: Approved style is required on this task. The rules in chapter 6 (page 59) apply. Be careful with your spacing, indenting, and choice of variable names.

There is a chapter in the textbook, probably about chapter 26, probably about page 142, that talks about string comparisons. They are basically like numeric comparisons, but use different operators.

Literal strings like Davis and Dodson should be enclosed in quote marks so they are not misunderstood. See section 6.9 (page 65) for details.

Sample Execution:

```
  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:  "Ditto would be on page 20."
 eof  (end of output)
```

7.12 g61: While Loop

- Discussed: Mon, May 11
- 11pt Due Date: Mon, May 11, 23:59.
- 10pt Due Date: Wed, May 13, 23:59.
- 7pt Due Date: Wed, Jun 10, 23:59.
- Grading Label: **g61**

GradeBot: This is a GradeBot task. If your program runs correctly and has proper indenting, GradeBot will send it to me for you. The general rules and explanations in section 4.3 (page 46) apply, including the required comment line.

`# cis101 g61 lastname, firstname` is the required comment line. There may be customizations that are different for each student based on this comment line. Your wordings may differ from study guide examples.

GradeBot is at `http://gradebot.tk/` and `http://gbot.dc.is2.byuh.edu/`.

Style: Approved style is required on this task. The rules in chapter 6 (page 59) apply. Be careful with your spacing, indenting, and choice of variable names.

Note: This assignment must pass GradeBot and then I will visually inspect it to see if the right kind of looping was used. Then I will confirm the points or do a throw-back.

There are chapters in the textbook, probably about chapters 31 and 32, probably starting about page 166, that talks about loops and while loops.

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 you should get as close as possible to the ending point without going beyond it.

Sample Execution:

```
  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"
12:  "Done!"
eof  (end of output)
```

7.13 o6F: Farm

- Discussed: Wed, May 13
- 11pt Due Date: Wed, May 13, 23:59.
- 10pt Due Date: Fri, May 15, 23:59.
- 7pt Due Date: Wed, Jun 10, 23:59.
- Grading Label: **o6F**

Online: 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 4.1 (page 42) apply. To request a regrade, see 3.4 (page 40).

There is a chapter in the textbook, probably about chapter 31, probably about page 163, that talks about loops (repeated actions).

Task: Print pictures of plants. Use a loop.

There is a good chance that many students will create programs that have an infinite loop. This will greatly slow down the IS2 machine.

THEREFORE: TEST YOUR PROGRAM AT YOUR DESKTOP BEFORE SAVING AND TESTING IT ONLINE. The input will be name=value where name is the name of your input field and value is the number that is keyed into that field.

Requirements:

Autofocus: Your program must display one (numeric) input field. Use autofocus (required!) to place the cursor in that field.

Blank: Blank out the numeric field between runs. Do not carry forward the latest entry.

Submit: Your program must have a submit button. When the submit button is pressed, the screen should be redrawn, followed by "n" pictures of your crop, where "n" is the number that was keyed into the input field.

Horizontal: The pictures must be in a row, not in a column. (If the row gets too long, it should automatically wrap to the next line. That is okay.)

Titles: Each picture must have a "title" that says "k of n" where k is the count, starting at 1, and n is the number being drawn. Example: if 7 are being drawn, the first would have a title "1 of 7".

Image Size: The display size of your images should be exactly 100 px wide and probably 100 px tall. You can specify width=100 if you want.

Max Stated: Your loop must not exceed some pre-determined limit of iterations. You must pick a limit between 11 and 19. You must clearly state what your limit is.

Max Enforced: If the user request is larger than your limit, you must complain and your loop must use your limit instead.

Include your name in the title and in an h1 at the top of the page. This gets you fame and glory. It also makes grading easier.

Suggested but not Required:

(a) Use the olin subroutine provided previously.

(b) Pick something amusing for your crop and for the name of your garden.

7.14 g46: Cheese

- Discussed: Fri, May 15
- 11pt Due Date: Fri, May 15, 23:59.
- 10pt Due Date: Mon, May 18, 23:59.
- 7pt Due Date: Wed, Jun 10, 23:59.
- Grading Label: **g46**

GradeBot: This is a GradeBot task. If your program runs correctly and has proper indenting, GradeBot will send it to me for you. The general rules and explanations in section 4.3 (page 46) apply, including the required comment line.

`# cis101 g46 lastname, firstname` is the required comment line. There may be customizations that are different for each student based on this comment line. Your wordings may differ from study guide examples.

GradeBot is at `http://gradebot.tk/` and `http://gbot.dc.is2.byuh.edu/`.

Style: Approved style is required on this task. The rules in chapter 6 (page 59) apply. Be careful with your spacing, indenting, and choice of variable names.

There is a chapter in the textbook, probably about chapter 20, probably about page 112, that talks about numeric comparisons.

Task: See how many packages of cheese should be purchased.

There are two ways to approach this task. One way is to check the remainder after division. If it is non-zero, add one more package.

The other way involves a clever use of int together with adding and subtracting one. No if statements are involved, but it is rather on the clever side of things.

7.15 g47: Appointment

- Discussed: Mon, May 18
- 11pt Due Date: Mon, May 18, 23:59.
- 10pt Due Date: Wed, May 20, 23:59.
- 7pt Due Date: Wed, Jun 10, 23:59.
- Grading Label: **g47**

GradeBot: This is a GradeBot task. If your program runs correctly and has proper indenting, GradeBot will send it to me for you. The general rules and explanations in section 4.3 (page 46) apply, including the required comment line.

`# cis101 g47 lastname, firstname` is the required comment line. There may be customizations that are different for each student based on this comment line. Your wordings may differ from study guide examples.

GradeBot is at `http://gradebot.tk/` and `http://gbot.dc.is2.byuh.edu/`.

Style: Approved style is required on this task. The rules in chapter 6 (page 59) apply. Be careful with your spacing, indenting, and choice of variable names.

There is a chapter in the textbook, probably about chapter 20, probably about page 112, that talks about numeric comparisons.

Task: See whether the person will be late for their appointment.

The difficulty in this task is that times are expressed in 12-hour format rather than the more convenient 24-hour format. A time of 1 comes later in the day than a time of 11, for instance.

7.16 o6D: Multi Dice

- Discussed: Mon, May 18
- 11pt Due Date: Mon, May 18, 23:59.
- 10pt Due Date: Wed, May 20, 23:59.
- 7pt Due Date: Wed, Jun 10, 23:59.
- Grading Label: **o6D**

Online: 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 4.1 (page 42) apply. To request a regrade, see 3.4 (page 40).

There is a chapter in the textbook, probably about chapter 31, probably about page 163, that talks about loops (repeated actions).

Summary: Make a web program that rolls the number of dice selected. Each die is rolled randomly, independent of the others.

See task oD and task oF for how-to information.

Requirements:

(a) **ID:** Include your name in the title and in an h1 at the top of the page.

(b) **Autofocus/Blank:** Your program must display and accept a (numeric) input field. Use autofocus (required!) to place the cursor in that field. Blank out the numeric field between runs. Do not carry forward the latest entry.

(c) **Max:** Near your input field, you must say what your maximum number of dice is. You can pick any maximum between 10 and 100. If the user request is larger than your limit, your loop must use your limit instead and print a suitable warning message.

(d) **Submit:** Your program must have a submit button. When the submit button is pressed, the screen should be redrawn, complete with the blank for numeric input, followed by "n" pictures of dice, where "n" is the number that was keyed into the input field. No particular title is required.

Instead of using `<img>` images of dice, you are welcome to use the HTML codes for dice.

`&#x2680;` is the html code for die face 1.
`&#x2681;` is the html code for die face 2.
`&#x2682;` is the html code for die face 3.
`&#x2683;` is the html code for die face 4.

`&#x2684;` is the html code for die face 5.
`&#x2685;` is the html code for die face 6.

(e) **Image Size:** Each image should have a displayed size that is large enough to see clearly, but no larger than 100px wide and 100px tall. You can use width= and height= if you like, or you can resize your images.

(f) **Alternatives:** Instead of normal, six-sided dice you can use something else (maybe slot machine images or playing cards) but it must have at least four alternatives from which one is selected.

7.17 g73: Boring

- Discussed: Wed, May 20
- 11pt Due Date: Wed, May 20, 23:59.
- 10pt Due Date: Wed, May 27, 23:59.
- 7pt Due Date: Wed, Jun 10, 23:59.
- Grading Label: **g73**

GradeBot: This is a GradeBot task. If your program runs correctly and has proper indenting, GradeBot will send it to me for you. The general rules and explanations in section 4.3 (page 46) apply, including the required comment line.

`# cis101 g73 lastname, firstname` is the required comment line. There may be customizations that are different for each student based on this comment line. Your wordings may differ from study guide examples.

GradeBot is at `http://gradebot.tk/` and `http://gbot.dc.is2.byuh.edu/`.

Your task is to listen to a list of foods and report which ones are boring. It is boring if you have had it before.

Our purpose is to give you training in working with arrays.

Each time around your outer loop, ask for a food. Then use an inner loop to check the list of foods already encountered. See if it is boring.

For style, I will care about the variable names you pick. They should be helpful and not ambiguous. Also, get the indenting right.

7.18 o6M: Maze

- Discussed: Wed, May 20
- 11pt Due Date: Wed, May 20, 23:59.
- 10pt Due Date: Wed, May 27, 23:59.
- 7pt Due Date: Wed, Jun 10, 23:59.
- Grading Label: **o6M**

Online: 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 4.1 (page 42) apply. To request a regrade, see 3.4 (page 40).

There is a chapter in the textbook, probably about chapter 31, probably about page 163, that talks about loops (repeated actions).

Summary: Make a web program that builds a maze of the dimensions requested. Use a nested loop. Each maze wall is constructed randomly, independent of the others.

Requirements:

(a) **Identity:** Include your name in the page title and in an h1 at the top of the page, or in a statement at the bottom of your page.

(b) **Two Fields:** Your program must display and accept two (numeric) input fields, one for width (number of columns) and one for height (number of rows). They must be clearly labeled.

(c) **Autofocus:** Use autofocus (required!) to place the cursor in the width field.

(d) **Sticky/Defaults:** Remember the numeric field values between runs. Default them to the maximum values you will accept. Draw your initial maze using those defaults.

(e) **Submit:** Your program must have a submit button.

(f) **Max Stated:** Near your input fields, you must say what your maximum number of rows and columns is. You can pick any maximum between 10 and 100, but the resulting maze must fit on a normal computer screen (1000 px wide).

(g) **Over Max:** If any user request is larger than your limit, your loop must use your limit instead and a suitable warning should be printed.

(h) **Walls:** The walls should be randomly either / or with a 50-50 probability. You can use the actual / and glyphs or you can find something else (like `&#x2571;` and `&#x2572;`), or draw an image and use it. You can choose to have more possibilities, but make sure the resulting maze looks at least as good as one with just the two options.

(i) **Spacing:** Control the font size and the line height (1em is recommended). Otherwise it will not look like a maze. You can use something like this in your style section.

```
div { font-size: 200%; line-height: 1em; }
```

7.19 g45: Afford

- Discussed: Wed, May 27
- 11pt Due Date: Wed, May 27, 23:59.
- 10pt Due Date: Mon, Jun 01, 23:59.
- 7pt Due Date: Wed, Jun 10, 23:59.
- Grading Label: **g45**

GradeBot: This is a GradeBot task. If your program runs correctly and has proper indenting, GradeBot will send it to me for you. The general rules and explanations in section 4.3 (page 46) apply, including the required comment line.

`# cis101 g45 lastname, firstname` is the required comment line. There may be customizations that are different for each student based on this comment line. Your wordings may differ from study guide examples.

GradeBot is at `http://gradebot.tk/` and `http://gbot.dc.is2.byuh.edu/`.

Warning: This one is a bit tricky. Read it carefully and think about it carefully. It sounds easy, but it is actually somewhat difficult.

Objective: Learn how to use if/else skillfully.

Summary: Consider two gifts and the money you have available. Tell which gifts, if any, to purchase.

Task: You are shopping for wedding gifts for a good friend. They have registered their wants on 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.

For cases that are not covered by these instructions, GradeBot will tell you what it wants you to say in each case.

Sample Execution:

```
 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)
```

7.20 o7T: Time

- Discussed: Wed, May 27
- 11pt Due Date: Wed, May 27, 23:59.
- 10pt Due Date: Mon, Jun 01, 23:59.
- 7pt Due Date: Wed, Jun 10, 23:59.
- Grading Label: **o7T**

Online: 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 4.1 (page 42) apply. To request a regrade, see 3.4 (page 40).

There is a chapter in the textbook, probably about chapter 41, probably about page 206, that talks about arrays and localtime.

Task: Create a dynamic webpage that shows the current date and time.

name: The webpage must include your name, either in the h1 or at the bottom of the page.

hms: It must show the current hours:minutes:seconds.

mmss: minutes and seconds must be formatted as two-digit numbers. See "formatted printing" in the index of the textbook. You want printf or sprintf with the zero-fill option: `%02d`.

dmy: It must show the current day of month, month, and year, with month as a word, not a number, and year as a four-digit number.

dow: It must show the current day of the week as a word.

(Note: There is another way to get the date and time. It is pre-formatted string like Fri Apr 16 12:34:56 2014. This will not be accepted. You must use the array version of localtime.)

As an example, you can look at my o7T webpage. You are welcome and encouraged to decorate your page, but you are not required to do so.

For assistance, you can look in the textbook index for "localtime." Or look up "gmtime" (Greenwich Mean Time). GMT is also called UTC.

You can do a Google search for "perl localtime" on the web.

The following suggestion might be helpful in working with the various arrays. It is intentionally incomplete.

```
( $s, $m, $h, $d, $mo, $y, ... ) = localtime;
```

```
$y += 1900; # correct the year
@dow = ( "Sun", "Mon", ..., "Sat" );
@mmm = ( ... );
```

7.21 g78: 24H to AM/PM

- Discussed: Mon, Jun 01
- 11pt Due Date: Mon, Jun 01, 23:59.
- 10pt Due Date: Wed, Jun 03, 23:59.
- 7pt Due Date: Wed, Jun 10, 23:59.
- Grading Label: **g78**

GradeBot: This is a GradeBot task. If your program runs correctly and has proper indenting, GradeBot will send it to me for you. The general rules and explanations in section 4.3 (page 46) apply, including the required comment line.

`# cis101 g78 lastname, firstname` is the required comment line. There may be customizations that are different for each student based on this comment line. Your wordings may differ from study guide examples.

GradeBot is at `http://gradebot.tk/` and `http://gbot.dc.is2.byuh.edu/`.

Task: Convert time from 24-hour format to AM/PM format.

Time is provided as HH:MM AM or HH:MM PM. You can use "split" to isolate information you want. You can use "if" to handle AM versus PM. You can use "printf" to format the output line.

There are other approaches that could also be successful.

Sample Executions:

```
GradeBot would have engaged your program in this dialog:
note  GBot "# debug output lines are permitted"
  1:  "Please enter the time as HH:MM (24h format): " (no \n)
 in>   ............................................"0:34"
  2:  "0:34 is 12:34 AM"
 eof  (end of output)
```

7.22 o7F: Farm 2

- Discussed: Mon, Jun 01
- 12pt Due Date: Mon, Jun 01, 23:59.
- 11pt Due Date: Wed, Jun 03, 23:59.
- 10pt Due Date: Mon, Jun 08, 23:59.
- 7pt Due Date: Wed, Jun 10, 23:59.
- Grading Label: **o7F**

This one is a bit harder so we may spend more time on it.

Online Plus Code: 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 4.1 (page 42) apply, including email subject line but not program comment line.

`cis101 o7F lastname, firstname` is the required subject line. There is no required comment line because this is not a GradeBot task.

There is a unit in the textbook, probably unit 8, probably about page 217, that talks about subroutines.

Task: Similar to o6F (above) but using subroutines. And multiple crops. You are planting a farm. Ask for planting directions. Show the results.

Objective: Learn to use subroutines to avoid repetition of code.

Danger Area: Be careful to properly use local variables in your subroutines. It is easy to accidentally have a global variable.

I will read your code to verify that you used the proper structure in writing your program. In your email, include a link to your online program. I will test it online to see how well it works.

7.22.1 Main Program

Your main program must have the following lines, or something very similar.

```
@crops = ( "a", "b", "c", "d" ); # four or more crops
foreach $crop ( @crops ) { plant ( $crop ) } # ask how many
# print an appropriate submit button
foreach $crop ( @crops ) { harvest ( $crop ) } # show results
```

7.22.2 Rules About Crops

You must have **at least four crops, and at most six crops.** The crops do not have to be actual farming crops. They can be something funny or weird. Pokemon. Soldiers. Books. Whatever.

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

The text strings in @crops will need to be usable for (a) display labels, (b) input names, and (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.

7.22.3 Subroutine: max

You must have a subroutine named max that returns a number of your choice for how many of each crop can be grown. This number must be at least 10 and at most 20.

This number must not appear elsewhere in your program. It can only appear here in the max subroutine. If this number is changed in this one place, the whole program must respond properly without further changes.

Every time you need the max, you should call this subroutine to retrieve it, or get it from a variable where you have stored it.

7.22.4 Subroutine: plant

You must have a subroutine named plant that accepts one parameter which is the name of the crop that is available for planting.

You must not use any global variables. For this subroutine, it should be easy.

For the designated crop, display a blank into which a number can be entered. Use the placeholder option to show the name of the crop being requested. (The name of the crop comes from the subroutine's parameter list.) Do not make the number "sticky." Each blank must be empty each time the screen is presented.

Put `\n` between the input fields in the HTML (but not on the screen). This avoids having a really wide line that I would have to scroll horizontally to examine when I view your page source. If the line is longer than about 100 characters, look for a good place to split it into two lines.

7.22.5 Subroutine: harvest

You must have a subroutine named harvest that accepts one parameter which is the name of the crop that will be grown.

You must not use any global variables. For this subroutine, you will need to be careful.

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

If the quantity is larger than max, complain, and use the smaller number. Example: if max is 9 and the requested quantity is more than 9, just use 9.

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

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

Each image should be about 100px by 100px.

Do not repeat pieces of code if you can avoid it. Specifically, you should have exactly one place where you print the image of the crop, not more than that.

7.22.6 Subroutine: olin

Use the subroutine olin that we previously introduced to find out what inputs were provided to your program. (This subroutine does have one global variable.)

7.23 g65: Box

- Discussed: Wed, Jun 03
- 11pt Due Date: Wed, Jun 03, 23:59.
- 10pt Due Date: Mon, Jun 08, 23:59.
- 7pt Due Date: Wed, Jun 10, 23:59.
- Grading Label: **g65**

GradeBot: This is a GradeBot task. If your program runs correctly and has proper indenting, GradeBot will send it to me for you. The general rules and explanations in section 4.3 (page 46) apply, including the required comment line.

`# cis101 g65 lastname, firstname` is the required comment line. There may be customizations that are different for each student based on this comment line. Your wordings may differ from study guide examples.

GradeBot is at `http://gradebot.tk/` and `http://gbot.dc.is2.byuh.edu/`.

Objective: Do nested loops.

Summary: Given the size of the box, draw a solid box using stars.

Notice that there are spaces between the stars (but not after the last star) to make the box look more square. This also makes the task slightly harder.

Sample Execution:

```
GradeBot would have engaged your program in this dialog:
note  GBot "# debug output lines are permitted"
  1:  "What size box? " (no \n)
 in>   ..............."2"
  2:  "* *"
  3:  "* *"
  4:  "Done!"
 eof  (end of output)
```

7.24 o6H: High Low

- Discussed: Wed, Jun 03
- 11pt Due Date: Wed, Jun 03, 23:59.
- 10pt Due Date: Mon, Jun 08, 23:59.
- 7pt Due Date: Wed, Jun 10, 23:59.
- Grading Label: **o6H**

Online: 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 4.1 (page 42) apply. To request a regrade, see 3.4 (page 40).

Summary: Program the high-low guessing game to run online. This requires a hidden field to hold the number being guessed.

I will automatically grade all programs at the 12pt, 11pt, and 10pt due dates. If you need your program graded after that, you should request it by email. Or you can have me grade it in class.

Requirement: It must include the usual things plus (a) an autofocus field which is blank into which a guess is entered, (b) a hidden field with the number to be guessed, and (c) a submit button. (Not all browsers require a submit button but some do.)

Requirement: (d) When the program first starts, it must pick a number to be guessed, which must be between 1 and 100, inclusive. (e) If the guess is too low, it should say so. (f) If the guess is too high, it should say so. (g) If the guess is correct, it should say so and immediately (h) pick a new number to be guessed.

There will be NO loops in this program, and your only STDIN will be part of olin. (Look in the textbook index for “memo to self”.)

Advice: While developing and testing your program, it is convenient to make the secret number into a regular (text) input field instead of a hidden field so you can see what is going on. After you get your program working, change it into a hidden field.

You are welcome to steal the graphics from my own high-low program, or make your own, or do your program without graphics.

Optional Idea: Count how many guesses were made and make a clever comment based on the skill or luck of the player.

Reminder: You must use autofocus to position the cursor in the input field.

Reminder: When the answer is guessed, you must immediately pick a new answer, which will normally be different from the previous answer.

7.25 g43: Leap Year

- Discussed: Mon, Jun 08
- 11pt Due Date: Mon, Jun 08, 23:59.
- 10pt Due Date: Wed, Jun 10, 23:59.
- Grading Label: **g43**

GradeBot: This is a GradeBot task. If your program runs correctly and has proper indenting, GradeBot will send it to me for you. The general rules and explanations in section 4.3 (page 46) apply, including the required comment line.

`# cis101 g43 lastname, firstname` is the required comment line. There may be customizations that are different for each student based on this comment line. Your wordings may differ from study guide examples.

GradeBot is at `http://gradebot.tk/` and `http://gbot.dc.is2.byuh.edu/`.

Task: Tell whether a given 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. Unless if it is a multiple of 100; then it is not. Unless if it is a multiple of 400; then it is.

You can do this with a single if statement that uses ands and ors.

You can do this with an if, elsif, else approach.

You are also welcome to use some other approach.

Helpful hint: The remainder operator, also called modulus, is represented by the percent sign. `11 % 3` means the remainder when eleven is divided by three, and the answer is two. (Three goes into eleven three times, with a remainder of two.) See the textbook for more information. Look up "remainder" in the index.

Sample Executions:

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

GradeBot would have engaged your program in this dialog:
  1:  "What is the year? " (no \n)
 in>   .................."2012"
  2:  "2012 is a leap year."
 eof  (end of output)
```

7.26 g7A: Secret Codes

- Discussed: Mon, Jun 08
- 11pt Due Date: Mon, Jun 08, 23:59.
- 10pt Due Date: Wed, Jun 10, 23:59.
- Grading Label: **g7A**

GradeBot: This is a GradeBot task. If your program runs correctly and has proper indenting, GradeBot will send it to me for you. The general rules and explanations in section 4.3 (page 46) apply, including the required comment line.

`# cis101 g7A lastname, firstname` is the required comment line. There may be customizations that are different for each student based on this comment line. Your wordings may differ from study guide examples.

GradeBot is at `http://gradebot.tk/` and `http://gbot.dc.is2.byuh.edu/`.

Use reasonable style and proper indenting.

Purpose: To gain experience in working with arrays.

Task: We will encrypt and decrypt messages using a simple transposition cipher.

Basically, Julius Caesar encrypted messages by shifting the letters. A shift of 1 would turn A into B, and X into Y. A shift of 25 (or -1) would turn B into A, and Y into X. For this task, only letters are changed. Punctuation is not changed.

The recommended approach is to use split to get at the individual characters. Then convert each character into a number. Then do some math to get the new number. Then convert the new number back into a character.

Split has two parameters, the string to be split, and the boundary characters. If the boundary characters are `""` then split will return an array with each original character by itself.

Perl has a subroutine "ord" that converts a letter into a number. A becomes 65. B becomes 66. a becomes 97. b becomes 98.

Perl has a subroutine "chr" that converts a number into a letter. 65 becomes A. 66 becomes B. 97 becomes a. 98 becomes b.

The math involves adding the shift amount to the number, and then taking the remainder from dividing by 26. Thus, 27 becomes 1.

7.27 oJS: JavaScript

- Discussed: Wed, Jun 10
- 11pt Due Date: Wed, Jun 10, 23:59.
- Grading Label: **oJS**

Online: 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 4.1 (page 42) apply. To request a regrade, see 3.4 (page 40).

Task: Create an html webpage (a) properly linked to the student projects page. It must include (b) your name, (c) a JavaScript calculator similar to the one on my demo page, (d) autofocus into the first data field.

HTML: Notice that you will be creating a webpage only, not a CGI program.

Maintain: The demo calculator has two fields (A and B), and buttons for add, subtract, multiply, and divide. **You must keep these fields and buttons.**

Divide: On the demo calculator, the divide button fails when you divide by zero. **You must fix it so it displays a special error message,** not just “Infinity” like JavaScript would normally say.

Beyond: Go beyond the demo example. Add another button, like maybe square root, or $a^2 + b^2$, hopefully something useful.

Chapter 8

Exam Questions

There are 21 exam questions. This chapter talks about each one and helps you avoid common mistakes.

Proper style is important, and required in most cases. The exceptions are: (a) the first three questions I mostly ignore style, and (b) on the very last final exam I mostly ignore style.

Testing is important. If I receive a program that was clearly not tested, it will be marked wrong and not given a second chance.

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.

GradeBot has some exercises that are similar to the exam questions. They are listed as “GradeBot Examples”. They might provide useful practice as you prepare to take an exam.

In the case of GradeBot, only behavior and results are measured. GradeBot does not examine your code to see how you achieved your result.

In the case of the exam itself, the human grader does review your code to make sure you achieved your result in the required manner.

8.1 q1: String Basic

GradeBot Examples: g21.

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.

8.2 q2: Number Basic

GradeBot Examples: g30 series.

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.

8.3 q3: Number Story

GradeBot Examples: g30 series.

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.

8.4 q4: Number Decision

GradeBot Examples: g40 series.

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.

8.5 q5: Number Decision Story

GradeBot Examples: g40 series.

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.

8.6 q6: String Decision

GradeBot Examples: g50 series.

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 textbook.)

(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.

8.7 q7: String Bracket

GradeBot Examples: g50 series.

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 textbook.)

(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.

8.8 q8: Repeat While

GradeBot Examples: g60 series.

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.

8.9 q9: Repeat For

GradeBot Examples: g60 series.

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 an `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.

8.10 q10: Repeat Last

GradeBot Examples: g60 series.

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.

8.11 q11: Repeat Nested

GradeBot Examples: g60 series.

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.

8.12 q12: List Basic

GradeBot Examples: g70 series.

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.

You will have to know that shift and unshift affect the front (start) of the list, and push and pop affect the back (end) of the list.

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 (start)** of a list.

Example: `$x = shift @x;`

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

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

8.13 q13: List Loop

GradeBot Examples: g70 series.

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 $_ }`

8.14 q14: Array Basic

GradeBot Examples: g70 series.

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.

8.15 q15: Array Loop

GradeBot Examples: g70 series.

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] }`

8.16 q16: Array Split

GradeBot Examples: g70 series, specifically g74, g75, g76, g77, g78.

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)

8.17 q17: Array Join

GradeBot Examples: g70 series.

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)

8.18 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.

8.19 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.

8.20 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.

8.21 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 $abc;` - creates a local variable with `$abc` as its name.

Example: `my ( $abc );` - creates a local variable with `$abc` as its name.

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`.

8.22 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.

Appendix A

Credit for Catching Spelling Errors

I offer extra credit for reports of spelling and grammar errors in my formal communications, by which I mean written materials like syllabi, study guides, and text books as well as current portions of webpages. This is very helpful to me in correcting spelling mistakes. And it sometimes gets my students to read my materials more carefully.

Using Tools: This has gotten to be sort of a game at times, which makes it fun. We can get into Grammar Nazi mode and be picky, picky, picky. Students will cut and paste my words into a document and then run a spelling checker or grammar checker. Or they will directly open the PDF in a spelling or grammar checker.

Tools Break: You are welcome to do this, but you should be aware that spelling and grammar checkers work by a simplified set of rules compared to real life. If there are two spellings for a word, the spelling checker will commonly only accept one and will reject the other. This does not make the other wrong.

How To Tell Me: To get this credit, send me an email with details of the error (or supposed error) that you have found.

What To Tell Me: Provide enough context that I can easily find the error. Maybe copy and paste the whole paragraph where the error is. I give less credit if you just send me the page number and the wrong word. It is too much like looking for a needle in a haystack. Give me context.

Warning: Language Changes: The truth about English, and probably all languages, is that language changes over time. New words are created. New spellings are accepted. New grammar happens. And old grammar is resurrected.

Style Guides: I generally follow the accepted practices as shown in style guides such as the Chicago Manual of Style. But I take exception to certain things like those that are noted below. For things that I have considered and listed below, even though they may show up with a checker, I do not consider them to be incorrect.

My Rules: My rules are (a) is it commonly done? (b) is it ambiguous? (c) is it pretty? These are the same rules used by grammarians, but our decisions in any given case may be different.

Here is a list of things that have been previously considered.

webpage: Modern usage is split between making this two words, "web page," and one word, "webpage." I have decided to go with the one word version.

website: Modern usage is split between making this two words, "web site," and one word, "website." I have decided to go with the one word version.

themself: Modern usage has tended away from gender-specific words like himself in favor of gender-neutral words. I have migrated from him and her to "singular" **them** as my solution of choice to the gender-neutral dictates of modern political correctness. Some dictionaries do not recognize themself as a word, and instead suggest themselves. For plural them, this would be correct, but for singular them, themself is correct and is documented to have been used as far back in time as the 1400s.

vs: - Should it have a dot? The usage argument is that in British writing, abbreviations are dotted when the final letters have been dropped, but not when the intermediate letters have been dropped. Versus removes intermediate letters. American usage may differ. I do not put a dot after it. I don't like how it looks with a dot. It is a conscious decision, not an error.

zeros: versus zeroes: Both are considered correct. Google says that zeros is more commonly used.

Ambiguous Plurals: The plural of 15 is 15s, not 15's. Using an apostrophe generally indicates possession, but people do commonly (and incorrectly) use an apostrophe for plurals when without it the meaning seems less clear. My choice when making a plural that would look ambiguous is to quote the string being pluralized. So, for me, the plural of (a) is (“a”s) rather than (a’s) or (as).

Ambiguous Quoted Punctuation: When should punctuation that is not part of a quote be moved inside the quote marks? Typesetters traditionally float a period (full stop) inside a trailing quote mark because it looks better that way. In computing, quote marks typically delimit strings that have special meaning, and putting punctuation inside the marks changes the meaning of the string. I usually float punctuation if it does not change the meaning of the thing quoted. Otherwise not.

Series Comma: Some people write a list of three things as (a, b and c), but others write it as (a, b, and c). I write it the second way. This is not an error. Both usages are correct, but I find the first usage to be ambiguous, so I almost always use the second form.

Index