CIS 101 Study Guide
Summer-1, 2012

Don Colton
Brigham Young University–Hawaii

April 21, 2012

This is a study guide for the CIS 101 class, Introduction to Programming, taught by Don Colton, Summer-1, 2012.

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

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

http://ipup.doncolton.com/

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

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

Contents

Chapter 1

How Points Are Earned

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

Effort: About 40% of the points you can earn are for effort.

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

Performance: About 60% of the points you can earn are for performance.

Performance includes answering questions on exams. It also includes doing a final project.

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

Passing the Class

This section is written mostly for my fellow teachers. Students can listen in.

What exactly is enough to pass the class? Each teacher seems to develop his or her own ideas about what is important. Is effort enough when performance is lacking? I also teach advanced classes and I have come to believe strongly that giving a C or higher for effort is a really bad idea. It just delays the recognition of their failure by a semester. I waffle on D. If this class is prerequisite to anything else, a C- should mean the student has enough background to move on. But without the basic skills to go forward, you might as well give students an F.

Our department uses a philosophy that each course has an owner although other faculty may teach it. As I write this, I am currently the owner for the class that uses this book (CIS 101, Beginning Programming) but I do not personally teach every section. The owner is responsible for establishing the performance levels required of students. The department approves those levels. Individual teachers can decide day to day what they will do, but they cannot compromise the performance targets.

Basic Expectations

Our standard is that basic material must be mastered so later courses can build on it. This is not a survey course or a high-level overview like, say, Art Appreciation. It is a "do it" class like, say, Drawing.

Mastery is divided into five topics: basics, decisions, loops, arrays, and subroutines. The "secret sauce" is doing things online. Roughly speaking, by the end of the semester, students must master the first two topics to pass the class (with a D). They must master the first three to get a C. They must master all five to get a B. They must also demonstrate ability with some advanced material to earn an A. For my own purposes I translate this into a point scale where mastery becomes points and points become grades.

1 Basics: Students write correct programs that use standard input and output to get information into and out of the computer. Programs run from a Graphical User Interface (**GUI**) or from a Command Line Interface (**CLI**). Students demonstrate the ability to use normal (scalar) variables to do calculations such as inches to centimeters. Students use fundamental mathematical operators including add, subtract, multiply, divide, and parentheses. Students understand that statements are executed in order, one after an other, and that later statements can change the values of variables from what the earlier statements established. We introduce style rules including naming of variables and spacing of written programs.

2 Decisions (if/elsif/else): Students write correct programs that deal differently with alternate cases, such as whether to put AM or PM after the time, or whether a check will be honored or will bounce. This includes skill with **Boolean** operators (those yielding a **True** or **False** answer) such as comparatives (less than, greater than, equal to, not equal to) and conjunctives (and, or, xor). This also includes following style rules of block indentation and spacing to make complex programs more readable.

3 Repeated Actions (Loops): Students write correct programs that deal with repetition of actions, such as filling out a table. Style is also emphasized. Operators like `++` and `+=` are introduced. `next`, `last`, and `redo` are introduced.

4a Repeated Data (Arrays as Lists): Students write correct programs that deal with lists of information. The `foreach` loop is mastered. `push`, `pop`, `shift`, and `unshift` are mastered.

4b Repeated Data (Arrays by Index): Students write correct programs that deal with numbered tables of information. Indexing (`[1]` and `[-1]`) is mastered.

5 Organizing (Subroutines): Students write correct subroutines to better organize and structure their code. Local and global scope is understood. Return values are used. Positional parameters are used. Variable-length parameter lists are used (because it's fairly easy to include).

6 Online: Students use Regular Expressions to decode the information sent by the browser. They use the event-driven approach of creating programs that respond to buttons pressed and fields filled in on forms. They use html and cgi to interact with the outside world.

Proving Mastery

To prove mastery students write programs. This is done on the Final Exam. We do a number of programs of varying difficulty. Each program is matched up with learning objectives that will normally be demonstrated by successfully completing the program.

Grading is done by learning objective. If the required objective seems to be adequately satisfied, we give the student the point. If not, we don't.

Because students do not all learn at the same rate, we do not care when students demonstrate mastery as long as it is by the last day of class. The jury is still out on how much this simply invites students to procrastinate.

Out of 36 points, 26 are based on the final exam, 6 on in-class activities (helping each other build toy projects during class time), and 4 on a final project. This kind of takes the teeth out of homework assignments, which are worth zero. What do we do about that? An all-or-nothing final can be pretty scary. So we compromise by giving ...

The Early Final

About once a week we offer an actual final exam. The questions are different each time, but are basically the same or of the same difficulty. The rule is that if a student passes any learning objective on the exam, they don't have to prove that skill again. This gives them a reason to take the tests and to make progress. The entire final is probably too much to take on the last day of class anyway. Knowing that part of the final is completed has proven to be a good motivator.

Because each Early Final is actually the real final, all the normal rules apply. The exams are closely proctored and performance must be at a final exam level. Toward the start of the semester very few students pass anything. Toward the end many students are passing many things.

There are three side-effects worth mentioning here. (1) The grade book scores get updated as students improve. A midterm score is not locked in. It can be erased and rewritten. This is more work for the teacher. (2) Students do not lose hope prematurely. There is always the chance that "next week I will get it right." (3) Because students retake things until they are mastered, there is less need to give partial credit for inadequate performance.

The unit tests throughout the book give actual test questions that have been used in the Early Finals to assess mastery of learning objectives. Students are also allowed to keep a copy of the exam and the work they did. After all students have taken the exam, they are allowed to share their efforts with each other.

During the exams we allow the students to test their work by running it at their local machine. However, they are not allowed to use any notes or outside resources, including web pages. They are only allowed to test their programs by running them locally. If there are reference materials we wish to make available, we put them in the test itself.

Scoring: We grade programs "by hand," visually examining the student code. In addition to working, we expect student programs to demonstrate the requested programming style (indentation, spacing, comments, naming) to make the programs easy to read and understand.

Review: Early in the semester I use the projector to review Early Final answers submitted by students. Answers are kept anonymous while they are reviewed by the whole class immediately after the test, which is given the first ten to twenty minutes of class. This lets me see how the students are doing. It lets students see what good and bad answers look like. This also allows the faster students to keep moving and helps them understand how to tutor others.

Later in the semester I let students review the answers and scores from the whole class, anonymously, on their own time.

Advanced Expectations

Some advanced skills must be demonstrated to earn an A. This is not mastery like the basics. It is "synthesis." It involves creation of a working project. Ideally the project is something fun like a game or a service that students can share with their friends. Ideally students can inject their own creativity into their project.

Chapter 2

1B: String Basic

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."`

Chapter 3

2B: Number Basic

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.

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

Chapter 4

2S: Number Story

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.

Chapter 5

Style Requirements

As your programs become more complex, style becomes important.

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

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

Spacing

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

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

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

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

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

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

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

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

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

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

Mathematical Parentheses

The form `$x = ( something );` is not allowed because it is confusingly ambiguous.

Ambiguous: `$x = ( something );` - Perl may accept it but I do not.

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

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

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

Parentheses are also used in defining arrays.

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

There is an ambiguity that we wish to avoid.

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

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

One Statement Per Line

Each statement should be on its own line.

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

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

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

Blank Lines

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

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

Indenting

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

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

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

Within the block, indenting must be increased by two.

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

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

Names

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

Long descriptive names can be abbreviated and explained when used.

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

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

Use the Values Provided

Often a problem will specify certain numbers or strings that define how the program should run. If possible, use those exact same values in writing your program.

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, then use it 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`

Chapter 6

4D: Number Decision

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

The key things to demonstrate here are:

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

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

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

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

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

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

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

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

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

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

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

Chapter 7

4S: Number Decision Story

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

Chapter 8

5D: String Decision

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.

Chapter 9

5B: String Bracket

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

The key things to demonstrate here are:

(a) How to handle “clarinet through costly”.

(b) How to handle “a-j, k-o, p-z”.

(c) How to handle all possible capitalizations.

Chapter 10

6W: Repeat While

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 comparision like `( $x < 100 )`.

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

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

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

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

Chapter 11

6F: Repeat For

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

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

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

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

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

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

Chapter 12

6L: Repeat Last

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

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

`if ( condition ) { last }`

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

Chapter 13

6N: Repeat Nested

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

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

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

Chapter 14

7B: List Basic

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

The key things to demonstrate here are:

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

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

(b) An array can be modified.

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

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

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

Example: `$x = pop @x;`

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

Example: `$x = shift @x;`

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

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

Chapter 15

7L: List Loop

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

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

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

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

Chapter 16

8B: Array Basic

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

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

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

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

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

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

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

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

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

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

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

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

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

Chapter 17

8L: Array Loop

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

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

Example: `$size = @array;`

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

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

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

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

Chapter 18

8S: Array Split

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

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

Common mistake: `$x = split ...`

Chapter 19

8J: Array Join

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

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

Common mistake: `@x = join ...`

Chapter 20

Subroutine Basics

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

Subroutines are defined using the following syntax:

```
sub name { block }
```

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

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

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

Chapter 21

9R: Subroutine Return

(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 );` - strings should be quoted

Example: `return @x;`

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

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

Chapter 22

9P: Positional Parameter

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

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

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

Chapter 23

9G: Globals and Locals

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

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

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

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

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

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

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

Chapter 24

9V: Variable Number of Parameters

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

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

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

Chapter 25

Final Projects

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

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

Size

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

Trust

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

Invent a Project

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

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

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

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

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

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

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

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