The Repetition Control Structure
Computer Science I (with a Fortran main target language)

In the last set of notes, the introduction discussed the notion of control structures. Sequence was the first fundamental control structure and selection was second. The third fundamental control structure is called repetition, and it is the focus of this set of notes. Repetition happens when, as its name implies, we repeat some block of code multiple times. As you'll see, incorporating repetition into our algorithms gives us a lot of power. With very little work, we are able to process large amounts of information.

I. Motivating the Need for Loops

Consider the following block of pseudocode:

- display a line with 1 and $2^1$
- display a line with 2 and $2^2$
- display a line with 3 and $2^3$
- display a line with 4 and $2^4$
- display a line with 5 and $2^5$
- display a line with 6 and $2^6$
- display a line with 7 and $2^7$
- display a line with 8 and $2^8$
- display a line with 9 and $2^9$
- display a line with 10 and $2^{10}$

Some questions:

1. What's the big picture of what it does?

2. What do you observe about it?

3. What if we wanted to go further?

So, let's try to seek an improvement:

Let's begin by writing an abstract form of any of the lines in terms of a variable:

How does that variable change between lines?

Where does it start? _____ Where does it end? _____

We can put these ideas all together concisely into something we call a loop (where we assume that $i$ exists as an integer):
II. Loops in General

Loops are one of the fundamental implementations of the repetition control structure, and, at this level, what you should think of any time you need to repeat anything. We call this this kind of problem solving iterative, because each step of a loop's execution is called an iteration.

Many loops have a variable upon which execution depends called a loop control variable or LCV for short:

- In our initial example, the LCV was _____________.
- In some cases, the LCV will be a counter like this, but it doesn't have to be. Other variables - sometimes indeed plural - can get involved in controlling whether or not a loop executes at all or repeats.

In general, we say a loop consists of the following components:

- An initial condition, which is what is true before the loop executes its first iteration, or, almost equivalently, code to force this initial condition to be true.
- A loop test or guard, which is a Boolean expression that is checked before each iteration. The loop test must be true for an iteration to execute. If the loop test is false, the loop does not execute any more and control goes to the code that follows the loop. It guards access to what's inside the loop, hence the term "guard."
- A loop body, which is whatever it is we want to repeat. It could be a line of code or it could be multiple. It could include other control structures. Often, but not always, it's in terms of the LCV.
- An update step, which is a command that changes an LCV between iterations of a loop. Sometimes this is subtle and sometimes it's obvious. Sometimes it's part of how a language implements a particular looping structure and sometimes it's mixed in among the loop body. But, regardless, it must introduce the chance that changes in the LCV could result in the guard being false (bringing the loop to termination).

Let's identify where each of these components appears in the loop we derived in the introductory problem:

Initial Condition:

Loop Test:

Loop Body:

Update Step:

All loops fall into one of two major categories:

- **Determinate loops** are loops where we know exactly how many iterations the loop will run
  _______________________________________________________
- **Indeterminate loops** are loops where __________________________________________________
  ___________________________________________________

Examples:
We'll look at both classes of loops in turn, along with specific programming language syntax for how to implement them.

### III. A Determinate Counting Loop

Okay, so we said for a loop to be determinate, we would need to know exactly how many iterations that loop would run before we begin execution of it. So it stands to reason that all we need to do to implement determinate loops is to keep track of how many iterations. Then, our LCV is a counter that tells what number of iteration we're on.

Since our LCV is a count of the iteration number, it **MUST** be an integer. (It is fundamentally flawed to use any other data type. Some languages and compilers enforce this strictly, but even if you get an opportunity to use a floating-point variable as a loop counter, don't. Computers are very good at comparing integers, but not as good with floating-point values. Roundoff error can creep in and you have funny things happening when you think you want to get as far as 3.5, but the value 3.49999 is stored and you fall short by an iteration.)

Let's not worry about what a loop body might do and instead just focus on the mechanics of counting for now. First, let's map the other three fundamental components of loops to loops that are counter-controlled:

- **Initial condition:**
- **Loop test:**
- **Update step:**

We could concisely summarize the idea of such a loop via the following line of pseudocode:

```plaintext
for i = start to stop by step
```

or,

```plaintext
for i = start to stop when step is understood to be 1
for i = start downto stop by step how we sometimes write it when we count backwards
```

Let's translate this to general compile-able code:

Let's look at a bunch of examples and translate a few to code:

<table>
<thead>
<tr>
<th>Pseudocode of Heading</th>
<th>i Values for Which Body Executes</th>
<th>Ending i Value</th>
<th>Code Translation</th>
</tr>
</thead>
<tbody>
<tr>
<td>for i = 1 to 5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>for i = 2 to 10 by 2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>for i = 0 to 9 by 2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>for i = 100 to 50 by -10 for i = 100 downto 50 by 10</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>for i = 30 to 15 by -6 for i = 30 downto 15 by 6</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Some important notes:

- The last iteration for which the loop body executes is always closer to the starting value in the direction of the step. The loop body \textit{never} executes for an LCV value beyond the stopping value.
- The LCV always has an ending value beyond the stopping value. Whether that is greater or less than the stopping value depends on whether the loop is counting up or down.
- Again, while you can count by 2, you should never try to count by \(1/2\). The step must always be a \textbf{whole number}.

IV. Examples with Determinate Counting Loops

Let's look at several examples.

Example 1: Using compile-able code, write a loop to display the integers from 1 to 10.

Example 2: Using compile-able code, write a loop to print out every 10th year from 1890 to 2020.

Example 3: Using compile-able code, write a loop to print out every other letter of the alphabet from \(A\) to \(K\).

Summations are very-closely related to determinate loops. The parts of the loop correspond closely to Sigma notation:

- The update step is understood to be incrementing by 1.
- What's below the Sigma is the initial condition.
- What's above the Sigma is the the stopping value.
- What's being summed would be processed in the loop body.

Example 4: Using compile-able code, compute the following sum:

\[
\sum_{i=1}^{20} 2^i + 4
\]
Example 5: Suppose a constant \( N \) is defined to be a number of data points. Write code to prompt a user to input \( N \) numbers and compute their average. Assume you have floating-point variables \( cur, sum, \) and \( mean \) defined as a current data point, the sum of all data points entered, and the mean of all data points entered.

Note: This full program is also available on the course site.

V. Nesting with Determinate Loops

We can nest control structures.

Here's an example of some pseudocode that nests a selection structure inside a loop:

```plaintext
for i = 0 to 15
    if i mod 2 = 0
        display i
```

That's equivalent to this loop:

```plaintext
for i = 0 to 15 by 2
    display i
```

But maybe we want to put line breaks cleanly after every 10th number:

```plaintext
for i = 1 to 100
    display i in a field of width 6
    if i mod 10 = 0
        print a line break
```

And, of course, we can also nest loops within loops:

```plaintext
for i = 0 to 5
    sum = 0
    for j = 1 to 30 by 10
        sum = sum*i + j
    display sum
```

What does it do?
VI. Moving toward General Loops

Recall the first loop we saw. Here's the pseudocode:

```plaintext
for i = 1 to 10
    display a line with i and 2^i
```

Let's write another loop that does the same thing:

Pseudocode: "Real" Code:

Let's identify where all the components of a loop show up:

- Initial condition
- Loop test
- Update step
- Loop body

VII. General Loops: Posttests vs. Pretests and Execution

It turns out the kind of looping structures we saw above can be used to solve any kind of looping problem. While we used them above for a determinate situation, they're better for indeterminate situations.

Here's a general form of a general looping structure in pseudocode:

```plaintext
initial-condition
while loop-test
    loop-body
    update-step
end
```

Here's how a few different languages implement it:

<table>
<thead>
<tr>
<th>Language</th>
<th>Pseudocode</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fortran</td>
<td><code>do</code></td>
</tr>
<tr>
<td></td>
<td><code>if (.not. loop-test) exit</code></td>
</tr>
<tr>
<td></td>
<td><code>loop-body</code></td>
</tr>
<tr>
<td></td>
<td><code>update-step</code></td>
</tr>
<tr>
<td></td>
<td><code>end do</code></td>
</tr>
<tr>
<td>C++/Java</td>
<td><code>initial-condition</code></td>
</tr>
<tr>
<td></td>
<td><code>while (loop-test)</code></td>
</tr>
<tr>
<td></td>
<td><code>{</code></td>
</tr>
<tr>
<td></td>
<td><code>loop-body</code></td>
</tr>
<tr>
<td></td>
<td><code>update-step</code></td>
</tr>
<tr>
<td></td>
<td><code>end</code></td>
</tr>
<tr>
<td>Matlab</td>
<td><code>initial-condition</code></td>
</tr>
<tr>
<td></td>
<td><code>while loop-test</code></td>
</tr>
<tr>
<td></td>
<td><code>loop-body</code></td>
</tr>
<tr>
<td></td>
<td><code>update-step</code></td>
</tr>
<tr>
<td></td>
<td><code>end</code></td>
</tr>
<tr>
<td>Python:</td>
<td><code>initial-condition</code></td>
</tr>
<tr>
<td></td>
<td><code>while loop-test:</code></td>
</tr>
<tr>
<td></td>
<td><code>loop-body</code></td>
</tr>
<tr>
<td></td>
<td><code>update-step</code></td>
</tr>
</tbody>
</table>

Notice, first, that, there's a lot in common here, and it makes a point I want to make: You're not learning a programming language, but you're learning how to program. If you learn one imperative programming language, you're not too far off from being able to pick up other imperative languages. If someone asks you to look at or edit code in an unfamiliar language some day, don't be scared off by that. If you pay attention to the lessons of this course, you're probably more equipped than you think you are.
Also, notice that Fortran's the weird one here: The condition it checks is the condition to stop the loop, rather than the condition to continue the loop, like most languages do.

Let's define another key term: All of the loops you saw check the condition, whether it's a condition to stop or continue, before the execution of the loop body. As such, they are pretest loops. This is in contrast to posttest loops, which check the condition after the execution of the loop body. In essence, with a posttest loop, the loop test is checking whether the next iteration ought to execute.

Not all languages provide the mechanism for a posttest loop, but the concept nicely solves some looping problems very naturally. Fortran and C++ and Java do:

<table>
<thead>
<tr>
<th>Fortran</th>
<th>C++/Java</th>
</tr>
</thead>
<tbody>
<tr>
<td>initial-condition (maybe)</td>
<td>initial-condition (maybe)</td>
</tr>
<tr>
<td>do</td>
<td>do</td>
</tr>
<tr>
<td>loop-body</td>
<td>loop-body</td>
</tr>
<tr>
<td>update-step</td>
<td>update-step</td>
</tr>
<tr>
<td>if (.not. loop-test) exit</td>
<td>if (loop-test)</td>
</tr>
<tr>
<td>end do</td>
<td>} while (loop-test)</td>
</tr>
</tbody>
</table>

The initial condition is a bit more hazy with posttest loops. Sometimes, it's not distinct and merely just part of the first iteration of the loop. In fact, in all of these general loops, the update step may be mixed somewhere in the middle of the loop body.

**CAUTION:** Some languages do allow leaving a loop from the middle of the loop body, but this is extremely poor practice that you should never do, even if a language or compiler allows it. Doing so is sloppy hacking at best, and certainly not good practice. In this course and in many others, doing so earns you an automatic zero on the assignment, so if this paragraph hasn't dissuaded you from such poor practices, that policy will.

Okay, so let's look at a few examples and look at the control flow. I'll present the examples in Fortran.

**Example 1:**

```fortran
j = 5
do
  if (j > 7) exit
  write(*, *) j
  j = j + 1
end do
```

**Example 2:**

```fortran
j = 10
do
  write(*, *) j
  j = j + 5
  if (j >= 30) exit
end do
```

**Question:** Were these examples truly indeterminate or were they determinate?
VIII. Using Sentinels

Indeterminate problems generally require some sort of outside intervention. Perhaps we're asking a user to input several values and letting the user choose when to quit. Perhaps we're letting a user choose from among some options over and over again and one of those is to quit. Perhaps we're processing all the data that's in a file. Perhaps we're checking for incoming information as long as some connection is live. (Situations with random numbers involved also fall into this category, and they are contained within the program, so there's one exception to the "outside world" idea.)

The first example most closely fits what we'll study here. Say we want to allow a user to input data points, but instead of saying "you must enter 20 values," we'll let the user decide when it's time to stop. But how would the user do that? We need some value that can be used to say it's time to stop. That value is called a sentinel.

**Example:** Assume we have integer variables value and count declared. Write code to allow a user to input an unknown number of values and simply count how many values were input.

**Question:** What challenges and limitations come into play with using sentinels this way?

---

*I'm out of content for this page, so here's a comic (xkcd, #1195). What does this have to do with looping? What went wrong?*
IX. A Problem

Work with a partner or two to solve this twist on an old problem:
Allow a user to input any number of inputs. Compute the mean of the inputs.

Assume you have floating-point variables \( cur \), \( sum \), and \( mean \) defined as a current data point, the sum of all data points entered, and the mean of all data points entered. Declare any other variables you need before you begin.

X. Menus

In building programs with user interfaces, we'll often use a menu, i.e. we'll present the user with a number of options and some way of selecting from those options, collect the user's option, and then respond appropriately to the options.

**Question:** What class of loop is this - determinate or indeterminate?

**Question:** How will we know to stop or continue the loop?

**Question:** Is a pretest or posttest better?

**Question:** How should we decide how to respond to each of the different options in the menu?
Here's an example of a very simple menu implemented in Fortran:

```fortran
integer :: choice ! user's menu choice: 1-4

do ! run menu system continuously

    write(*, '(a,$)') "Choose an upcoming break " ! display menu
    write(*, *) " to find out when it starts: "
    write(*, *) "  1. Thanksgiving"
    write(*, *) "  2. Winter Break"
    write(*, *) "  3. Spring Break"
    write(*, *) "  4. Quit"
    write(*, '(a, $)') "Enter the number of your choice: "
    read *, choice

    select case(choice) ! output break chosen
      case(1) ! Thanksgiving
          write(*, '(a)') "Start date: November 23"
      case(2) ! Winter Break
          write(*, '(a)') "Start date: December 21"
      case(3) ! Spring Break
          write(*, '(a)') "Start date: March 8"
      case(4) ! Quit (gracefully)
          write(*, '(a)') "Have a nice day"
      case default ! Error Handle
          write(*,'(a)') "Enter choice between 1 and 4"
    end select

    write(*,*) " " ! leave space between runs
    if(choice == 4) exit ! 4 means "quit"
end do
```

**XI. Improved Validation**

Earlier in the course, we studied a technique for validating inputs in the input phase of our program to ensure that we didn't do calculations on bad data or output meaningless results. Our old technique did let the user know what the problem was, but was limited in that it didn't give the user a chance to fix his/her mistake. We can do better with loops. Consider this strategy:

- Put a pair of prompt-then-input statements in a loop with a posttest.
- Keep this loop going until the user inputs valid data for that input.
- Display an error message inside the loop, conditionally.

In a sense, this strategy forces the user to enter valid input before it's possible to proceed. As it's impossible to proceed without having valid input, there's no need to store a Boolean flag for bad input either, so it's not only better, but also cleaner.

**Example:** Suppose you have integer variable `startValue` that must be positive. Input start value such that it's required to be positive.
Question: Why did we use the loop we did?

Practice: Your turn. Say we're using startValue and an analogous endValue as the first and the last rows of some data table we're displaying. Input and validate endValue.

XII. Choosing the Right Loop for the Job

We've talked about different classes of loops and their implementations throughout these notes.

Each construct a programming language provides has a reason for existing, and, while you can twist some tools to fit other scenarios (really, though, that's abuse of the tools), your code is cleanest and most natural when you use the right tools for the right job. Let's summarize the ideal application of loops via a decision tree here:

XII. Looping Pitfalls

Here's a pretty bad Fortran loop:

```fortran
count = 0

do
  if (count > 100) exit
    sum = sum + input
    write(*, '(a)') "Enter another input: "
    read *, input
  end do
```

What's wrong with it?
This gives rise to a few general closing tips:

- Always have an update step. It's easy to forget with indeterminate loops (and that's another good reason to choose the right loop for the job). I suggest you write it at the beginning.
- Check that you have all four elements of the loop, however subtle they may be: initial condition, stopping condition/loop test, update step, loop body.
- In looping problems, you may need to initialize variables other than the LCV before starting the loop. Make sure you plan out your algorithm carefully.
- Make sure the update step always brings the loop closer to termination.
- You can nest with any kind of loop.