16. Controlling Loops

Code Complete

Contents

"LOOP" IS AN INFORMAL TERM that refers to any kind of iterative control structure—any structure that causes a program to repeatedly execute a block of code.

Common loop types are `for`, `while`, and `do-while` in C++ and Java.

16.1 Selecting the Kind of Loop

16.1.1 When to Use a `while` Loop

- Novice programmers sometimes think that a while loop is continuously evaluated and that it terminates the instant the while condition becomes false, regardless of which statement in the loop is being executed (Curtis et al. 1986).
- Although it's not quite that flexible, a while loop is a flexible loop choice. If you don't know ahead of time exactly how many times you'll want the loop to iterate, use a `while` loop.

16.1.2 When to Use a `for` Loop

- A `for` loop is a good choice when you need a loop that executes a specified number of times. You can use for in C++, C, Java, Visual Basic, and most other languages.
- Use `for` loops for simple activities that don't require internal loop controls. Use them when the loop control involves simple increments or simple decrements.

- The point of a for loop is that you set it up at the top of the loop and then forget about it. You don't have to do anything inside the loop to control it. If you have a condition under which execution has to jump out of a loop, use a while loop instead.

- Likewise, don't explicitly change the index value of a for loop to force it to terminate. Use a while loop instead. The for loop is for simple uses. Most complicated looping tasks are better handled by a while loop.

16.1.3 When to Use a foreach Loop

- The `foreach` loop or its equivalent (foreach in C#, For-Each in Visual Basic, `For-In` in Python), is useful for performing an operation on each member of an array or other container.
- It has the advantage of eliminating loop-housekeeping arithmetic, and therefore eliminating any chance of errors in the loop housekeeping arithmetic.

16.2 Controlling the Loop

What can go wrong with a loop? Any answer would have to include at the very least:

- incorrect or omitted loop initialization,
- omitted initialization of accumulators or other variables related to the loop,
- improper nesting,
- incorrect termination of the loop,
- forgetting to increment a loop variable or incrementing the variable incorrectly,
- and indexing an array element from a loop index incorrectly.

You can forestall these problems by observing two practices:

1. First, minimize the number of factors that affect the loop. Simplify! Simplify! Simplify!
2. Second, treat the inside of the loop as if it were a routine—keep as much of the control as possible outside the loop. Explicitly state the conditions under which the body of the loop is to be executed. Don't make the reader look inside the loop to understand the loop control. Think of a loop as a black box

   C++ Example of Treating a Loop as a Black Box:
   What are the conditions under which this loop terminates?:

```
while ( !inputFile.EndOfFile() && moreDataAvailable ) {
        ...
}
```

Exiting Loops Early

- Many languages provide a means of causing a loop to terminate in some way other than completing the for or while condition. In this discussion, break is a generic term for **`break`** in C++, C, and Java.

- The break statement (or equivalent) causes a loop to terminate through the normal exit channel; the program resumes execution at the first statement following the loop.

- Consider using break statements rather than boolean flags in a while loop.
    - In some cases, adding boolean flags to a while loop to emulate exits from the body of the loop makes the loop hard to read.

- Sometimes you can remove several levels of indentation inside a loop and simplify loop control just by using a break instead of a series of if tests.
    - Putting multiple break conditions into separate statements and placing them near the code that produces the break can reduce nesting and make the loop more readable.

- Be wary of a loop with a lot of breaks scattered through it:
    - A loop's containing a lot of breaks can indicate unclear thinking about the structure of the loop or its role in the surrounding code. A proliferation of breaks raises the possibility that the loop could be more clearly expressed as a series of loops rather than as one loop with many exits.

- Use labeled `break` if your language supports it
    - Java supports use of labeled breaks

- Use `break` and `continue` only with caution
    - Use of break eliminates the possibility of treating a loop as a black box. Limiting yourself to only one statement to control a loop's exit condition is a powerful way to simplify your loops. Using a break forces the person reading your code to look inside the loop for an understanding of the loop control. That makes the loop more difficult to understand
    - Some computer scientists argue that they are a legitimate technique in structured programming; some argue that they are not. Because you don't know in general

whether continue and break are right or wrong, use them, but only with a fear that you might be wrong.

- Checking Endpoints
  - A single loop usually has three cases of interest: the first case, an arbitrarily selected middle case, and the last case. When you create a loop, mentally run through the first, middle, and last cases to make sure that the loop doesn't have any off-by-one errors.You can expect several benefits from mental simulations and hand calculations.
  - The mental discipline results in fewer errors during initial coding, in more rapid detection of errors during debugging, and in a better overall understanding of the program. The mental exercise means that you understand how your code works rather than guessing about it

Using Loop Variables

Here are some guidelines for using loop variables:

- Use ordinal or enumerated types for limits on both arrays and loops.
  Generally, loop counters should be integer values.

- Use meaningful variable names to make nested loops readable
  - Arrays are often indexed with the same variables that are used for loop indexes.
  - If you have a one-dimensional array, you might be able to get away with using i, j, or k to index it.
  - But if you have an array with two or more dimensions, you should use meaningful index names to clarify what you're doing.

Java Example of Bad Loop Variable Names

```
for ( int i = 0; i < numPayCodes; i++ ) {
      for ( int j = 0; j < 12; j++ ) {
            for ( int k = 0; k < numDivisions; k++ ) {
                  sum = sum + transaction[ j ][ i ][ k ];
            }
      }
 }
```

Java Example of Good Loop Variable Names

```
for ( int payCodeIdx = 0; payCodeIdx < numPayCodes; payCodeIdx++ ) {
   for (int month = 0; month < 12; month++ ) {
      for ( int divisionIdx = 0; divisionIdx < numDivisions;
            divisionIdx++ ) {
        sum = sum + transaction[ month ][ payCodeIdx ][ divisionIdx ];
      }
   }
}
```

- Limit the scope of loop-index variables to the loop itself
  Loop-index cross-talk and other uses of loop indexes outside their loops is such a significant problem.

How Long Should a Loop Be?

Loop length can be measured in lines of code or depth of nesting.

- Make your loops short enough to view all at once
  When you begin to appreciate the principle of writing simple code, however, you'll rarely write loops longer than 15 or 20 lines.

- Limit nesting to three levels
  - Studies have shown that the ability of programmers to comprehend a loop deteriorates significantly beyond three levels of nesting (Yourdon 1986a).
  - If you're going beyond that number of levels, make the loop shorter (conceptually) by breaking part of it into a routine or simplifying the control structure.
- Move loop innards of long loops into routines (mover las entrañas de un loop extenso a rutinas):
  If the loop is well designed, the code on the inside of a loop can often be moved into one or more routines that are called from within the loop.

- Make long loops especially clear: length adds complexity.

- If you write a short loop, you can use riskier control structures such as break and continue, multiple exits, complicated termination conditions, and so on.
- If you write a longer loop and feel any concern for your reader, you'll give the loop a single exit and make the exit condition unmistakably clear.

16.3 Creating Loops Easily—from the Inside Out

EJEMPLIFICARLO con el ejercicio de la TABLA DE MULTIPLICAR.

Here's the general process:

1. Start with one case. Code that case with literals.
2. Then indent it and replace the literals with loop indexes or computed expressions.
3. Put a loop around it.
4. Put another loop around that, if necessary, and replace more literals.
5. Continue the process as long as you have to.
6. When you finish, add all the necessary initializations.
7. Si ha usado un while, añade la sentencia que modifica la expresión de entrada al while.
8. Since you start at the simple case and work outward to generalize it, you might think of this as coding from the inside out.

16.4 Correspondence Between Loops and Arrays

Loops and arrays are often related. In many instances, a loop is created to perform an array manipulation, and loop counters correspond one-to-one with array indexes. For example, the Java for loop indexes below correspond to the array indexes:

Java Example of an Array Multiplication:

```
for ( int row = 0; row < maxRows; row++ ) {
      for ( int column = 0; column < maxCols; column++ ) {
         product[row][column] = a[row][column] * b[row][column];
      }
}
```

Some languages, provide powerful array operations that eliminate the need for loops like the one above. Here's an APL code fragment that performs the same operation:

APL Example of an Array Multiplication

```
Product <- a x b
```

The language you use to solve a problem substantially affects your solution.

Key Points

- Loops are complicated. Keeping them simple helps readers of your code.
- Techniques for keeping loops simple include avoiding exotic kinds of loops, minimizing nesting, making entries and exits clear, and keeping housekeeping code in one place.
- Loop indexes are subjected to a great deal of abuse. Name them clearly and use them for only one purpose.
- Think the loop through carefully to verify that it operates normally under each case and terminates under all possible conditions.

CHECKLIST: Loops

Loop Selection and Creation

- ❏ Is a while loop used instead of a for loop, if appropriate?
- ❏ Was the loop created from the inside out?

Entering the Loop

- ❏ Is the loop entered from the top?
- ❏ Is initialization code directly before the loop?
- ❏ If the loop is an infinite loop or an event loop, is it constructed cleanly
- ❏ rather than using a kludge such as for i = 1 to 9999?
- ❏ If the loop is a C++, C, or Java for loop, is the loop header reserved for
- ❏ loop-control code?

Inside the Loop

- ❏ Does the loop use { and } or their equivalent to prevent problems arising
- ❏ from improper modifications?
- ❏ Does the loop body have something in it? Is it nonempty?
- ❏ Are housekeeping chores grouped, at either the beginning or the end of the loop?
- ❏ Does the loop perform one and only one function—as a well-defined routine
- ❏ does?
- ❏ Is the loop short enough to view all at once?
- ❏ Is the loop nested to three levels or less?
- ❏ Have long loop contents been moved into their own routine?
- ❏ If the loop is long, is it especially clear?

Loop Indexes

- ❏ If the loop is a for loop, does the code inside it avoid monkeying with the
- ❏ loop index?
- ❏ Is a variable used to save important loop-index values rather than using the
- ❏ loop index outside the loop?
- ❏ Is the loop index an ordinal type or an enumerated type—not floating point?
- ❏ Does the loop index have a meaningful name?
- ❏ Does the loop avoid index cross talk?

Exiting the Loop

- ❏ Does the loop end under all possible conditions?
- ❏ Does the loop use safety counters—if you've instituted a safety-counter standard?
- ❏ Is the loop's termination condition obvious?
- ❏ If break or continue are used, are they correct?