Topics

- Iterative Statements
- Unconditional Branching
- Guarded Commands

Iterative Statements

- The repeated execution of a statement or compound statement is accomplished either by iteration or recursion
- General design issues for iteration control statements:
  1. How is iteration controlled?
  2. Where is the control mechanism in the loop?

Counter-Controlled Loops

- A counting iterative statement has a loop variable, and a means of specifying the *initial* and *terminal*, and *stepsize* values
- Design Issues:
  1. What are the type and scope of the loop variable?
  2. Should it be legal for the loop variable or loop parameters to be changed in the loop body, and if so, does the change affect loop control?
  3. Should the loop parameters be evaluated only once, or once for every iteration?
  4. What is the value of the loop variable after loop termination?

Counter-Controlled Loops: Examples

- C-based languages

```
for ([expr_1] ; [expr_2] ; [expr_3]) statement
```

  - The expressions can be whole statements, or even statement sequences, with the statements separated by commas
    - The value of a multiple-statement expression is the value of the last statement in the expression
    - If the second expression is absent, it is an infinite loop
- Design choices:
  - There is no explicit loop variable
  - Everything can be changed in the loop
  - The first expression is evaluated once, but the other two are evaluated with each iteration
  - It is legal to branch into the body of a for loop in C

Counter-Controlled Loops: Examples

- C++ differs from C in two ways:
  1. The control expression can also be Boolean
  2. The initial expression can include variable definitions (scope is from the definition to the end of the loop body)
- Java and C#
  - Differs from C++ in that the control expression must be Boolean

Counter-Controlled Loops: Examples

- Python

  `for` loop_variable **`in`** object**`:`**
  - loop body

  [**`else:`**
  - else clause]

  - The object is often a range, which is either a list of values in brackets (`[2, 4, 6]`), or a call to the **`range`** function, as in **`range`**`(5)`, which returns `0, 1, 2, 3, 4`
  - The loop variable takes on the values specified in the given range, one for each iteration
  - At loop termination, the loop variable has the last value that was assigned to it
  - The else clause, which is optional, is executed if the loop terminates normally

Counter-Controlled Loops: Examples

- F#
  - Because counters require variables, and functional languages do not have variables, counter-controlled loops must be simulated with recursive functions

```
let rec forLoop loopBody reps =
  if reps <= 0 then ()
  else
    loopBody()
    forLoop loopBody, (reps - 1)
```

  - This defines the recursive function `forLoop` with the parameters `loopBody` (a function that defines the loop's body) and the number of repetitions
  - `()` means do nothing and return nothing

Logically-Controlled Loops

- Repetition control is based on a Boolean expression
- Design issues:
  - Pretest or posttest?
  - Should the logically controlled loop be a separate statement?

Logically-Controlled Loops: Examples

- C and C++ have both pretest and posttest forms, in which the control expression can be arithmetic:

```
while (control_expr)
    loop body
```

```
do
    loop body
while (control_expr)
```

  - In both C and C++ it is legal to branch into the body of a logically-controlled loop

- Java is like C and C++, except the control expression must be Boolean (and the body can only be entered at the beginning -- Java has no `goto`

Logically-Controlled Loops: Examples

- F#
  - As with counter-controlled loops, logically-controlled loops can be simulated with recursive functions

```
let rec whileLoop test body =
   if test() then
     body()
     whileLoop test body
   else ()
```

  - This defines the recursive function `whileLoop` with parameters `test` and `body`, both functions. `test` defines the control expression

User-Located Loop Control Mechanisms

- Sometimes it is convenient for the programmers to decide a location for loop control (other than top or bottom of the loop)
- Simple design for single loops (e.g., `break`)
- Design issues for nested loops
  1. Should the conditional be part of the exit?
  2. Should control be transferable out of more than one loop?

User-Located Loop Control Mechanisms

- C , C++, Python, Ruby, and C# have unconditional unlabeled exits (`break`)
- Java and Perl have unconditional labeled exits (`break` in Java, `last` in Perl)
- C, C++, and Python have an unlabeled control statement, `continue`, that skips the remainder of the current iteration, but does not exit the loop
- Java and Perl have labeled versions of `continue`

Iteration Based on Data Structures

- The number of elements in a data structure controls loop iteration
- Control mechanism is a call to an *iterator* function that returns the next element in some chosen order, if there is one; else loop is terminate
- C's **`for`** can be used to build a user-defined iterator:

```
for (p=root; p==NULL; traverse(p)){
  ...
}
```

Iteration Based on Data Structures (continued)

- PHP
  - `current` points at one element of the array
  - `next` moves `current` to the next element
  - `reset` moves `current` to the first element

- Java 5.0 (uses `for`, although it is called foreach)

  For arrays and any other class that implements the `Iterable` interface, e.g., `ArrayList`

```
for (String myElement : myList) { … }
```

Iteration Based on Data Structures (continued)

- C# and F# (and the other .NET languages) have generic library classes, like Java 5.0 (for arrays, lists, stacks, and queues). Can iterate over these with the **`foreach`** statement. User-defined collections can implement the `IEnumerator` interface and also use **`foreach`**.

```
List<String> names = new List<String>();
names.Add("Bob");
names.Add("Carol");
names.Add("Ted");
foreach (Strings name in names)
    Console.WriteLine ("Name: {0}", name);
```

Iteration Based on Data Structures (continued)

- Ruby *blocks* are sequences of code, delimited by either braces or **`do`** and **`end`**
  - Blocks can be used with methods to create iterators
  - Predefined iterator methods (`times`, `each`, `upto`):

```
3.times {puts "Hey!"}
list.each {|value| puts value}
```

(`list` is an array; `value` is a block parameter)

```
1.upto(5) {|x| print x, " "}
```

Unconditional Branching

- Transfers execution control to a specified place in the program
- Represented one of the most heated debates in 1960's and 1970's
- Major concern: Readability
- Some languages do not support `goto` statement (e.g., Java)
- C# offers `goto` statement (can be used in `switch` statements)
- Loop exit statements are restricted and somewhat camouflaged `goto`'s

Guarded Commands

- Designed by Dijkstra
- Purpose: to support a new programming methodology that supported verification (correctness) during development
- Basic Idea: if the order of evaluation is not important, the program should not specify one

Selection Guarded Command

- Form

```
if <Boolean expr> -> <statement>
[] <Boolean expr> -> <statement>
 ...
[] <Boolean expr> -> <statement>
fi
```

- Semantics: when construct is reached,
  - Evaluate all Boolean expressions
  - If more than one are true, choose one non-deterministically
  - If none are true, it is a runtime error

Selection Guarded Command

```
if x >= y -> max := x
[] y >= x -> max := y
fi
```

Loop Guarded Command

- **Form**

  do <Boolean> -> <statement>

  [] <Boolean> -> <statement>

  ...

  [] <Boolean> -> <statement>

  od

- Semantics: for each iteration
  - Evaluate all Boolean expressions
  - If more than one are true, choose one non-deterministically; then start loop again
  - If none are true, exit loop

Loop Guarded Command

Given four integer variables, q1, q2, q3, and q4, rearrange the values of the four so that q1 ≤ q2 ≤ q3 ≤ q4.

```
do q1 > q2 -> temp := q1; q1 := q2; q2 := temp;
[] q2 > q3 -> temp := q2; q2 := q3; q3 := temp;
[] q3 > q4 -> temp := q3; q3 := q4; q4 := temp;
od
```

Guarded Commands: Rationale

- Connection between control statements and program verification is intimate
- Verification is impossible with `goto` statements
- Verification is possible with only selection and logical pretest loops
- Verification is relatively simple with only guarded commands

Assignments and others

- Reading assignment: Chapter 8
- Written assignment: assignment three (4%), due on February 15
- Exam one: February 22 (review changed to Feb. 17), Ch 1, 3-8.