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?

Iterative Statements: Examples

- FORTRAN 95 syntax
  ```fortran
  DO label var = start, finish [, stepsize]
  -- Stepsize can be any value but zero
  -- Parameters can be expressions
  . . .
  END Do [name]
  ```

  Design choices:
  1. Loop variable must be INTEGER
  2. The loop variable cannot be changed in the loop, but the parameters can; because they are evaluated only once, it does not affect loop control (Loop parameters are evaluated only once)
  3. Cannot branch into either of Fortran’s Do statements
Iterative Statements: Examples

- **Ada**
  ```for var in [reverse] discrete_range loop...
  end loop```
  - **Design choices:**
    1. Type of the loop variable is that of the **discrete range** (A discrete range is a sub-range of an integer or enumeration type).
    2. Loop variable does not exist outside the loop
    3. The loop variable cannot be changed in the loop, but the discrete range can; it does not affect loop control
    4. The discrete range is evaluated just once
    5. Cannot branch into the loop body

- **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:**
    1. There is no explicit loop variable
    2. Everything can be changed in the loop
    3. The first expression is evaluated once, but the other two are evaluated with each iteration

- **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

- **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 (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
  - The else clause, which is optional, is executed if the loop terminates normally
Logically-Controlled Loops

- Repetition control is based on a Boolean expression
- **Design issues:**
  - **Pretest** or **posttest**?
  - Should the logically controlled loop be a special case of the counting loop statement or 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 (ctrl_expr) do
  loop body
  loop body
  while (ctrl_expr)
  ```

- **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`)

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

break and continue

- 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

- Number of elements of in a data structure control 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 (cont'd)

- PHP
  - current points at one element of the array
  - next moves current to the next element
  - reset moves current to the first element
- Java
  - For any collection that implements the Iterator interface
  - next moves the pointer into the collection
  - hasNext is a predicate
  - remove deletes an element
- Perl has a built-in iterator for arrays and hashes, foreach

Iteration Based on Data Structures (cont'd)

- C#'s foreach statement iterates on the elements of arrays and other collections:
  Strings[] = strList = {"Bob", "Carol", "Ted"};
  foreach (Strings name in strList)
    Console.WriteLine ("Name: {0}", name);
  - The notation {0} indicates the position in the string to be displayed
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
• Basis for two linguistic mechanisms for concurrent programming (in CSP and Ada)
• **Basic Idea:** if the order of evaluation is not important, the program should not specify one

Selection Guarded Command

• **Form**
  
  ```
  if <Boolean exp> -> <statement>
  [] <Boolean exp> -> <statement>
  ... 
  [] <Boolean exp> -> <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

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
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

Summary

- Variety of statement-level structures
- Choice of control statements beyond selection and logical pretest loops is a trade-off between language size and writability
- Functional and logic programming languages are quite different control structures (not discussed here)