Chapter 8

Statement-Level Control Structures

Chapter 8

Statement-Level Control Structures

8.1 Introduction 330

- A control structure is a control statement and the statements whose execution it controls
  - Selection Statements
  - Iterative Statements
- There is only one design issue that is relevant to all of the selection and iteration control statements:
  - Should a control structure have multiple entries?

8.2 Selection Statements 332

- A selection statement provides the means of choosing between two or more paths of execution.
- Selection statement fall into two general categories:
  - Two-way selection
  - Multiple-way selection

8.2.1 Two-Way Selection Statements

- The general form of a two-way selector is as follows:

```
if control_expression
   then clause
   else clause
```

- Design issues
  - What is the form and type of the control expression?
  - How are the **`then`** and **`else`** clauses specified?
  - How should the meaning of nested selectors be specified?

- The control expression
  - Control expressions are specified in parenthesis if the `then` reserved word is not used to introduce the `then` clause, as in the C-based languages
  - In C89, which did not have a Boolean data type, **arithmetic** expressions were used as control expressions
  - In contemporary languages, such as Java and C#, **only Boolean** expressions can be used for control expressions

- Clause Form
  - In most contemporary languages, the `then` and `else` clauses either appear as single statements or compound statements.
  - C-based languages use **braces** to form compound statements.
  - One exception is **Perl**, in which all then and else clauses must be **compound** statements, even if they contain single statements
  - In Python and Ruby, clauses are statement sequences
  - Python uses **indentation** to define clauses

```
if x > y :
   x = y
   print " x was greater than y"
```

    - All statements equally indented are included in the compound statement. Notice that rather than `then`, a colon is used to introduce the `then` clause in the Python

- Nesting Selectors
  - In Java and contemporary languages, the static semantics of the language specify that the `else` clause is always paired with the **nearest** unpaired `then` clause

```
if (sum == 0)
    if (count == 0)
         result = 0;
else
   result = 1;
```

    - A rule, rather than a syntactic entity, is used to provide the disambiguation
    - So, in the example, the `else` clause would be the alternative to the second `then` clause
    - To force the alternative semantics in Java, a different syntactic form is required, in which the inner `if` is put in a compound, as in

```
if (sum == 0) {
    if (count == 0)
         result = 0;
}
else
   result = 1;
```

  - C, C++, and C# have the same problem as Java with selection statement nesting
  - Ruby, statement sequences as clauses:

```
if sum == 0 then
    if count == 0 then
         result = 0
    else
         result = 1
    end
end
```

  - Python, all statements uses indentation to define clauses

```
if sum == 0 :
    if count == 0 :
         result = 0
    else :
         result = 1
```

8.2.2 Multiple Selection Constructs

- The multiple selection construct allows the selection of one of any number of statements or statement groups.

- Design Issues
  - What is the form and type of the control expression?
  - How are the selectable segments specified?
  - Is execution flow through the structure restricted to include just a single selectable segment?
  - How are case values specified?
  - What is done about unrepresented expression values?

- C, C++, Java, and JavaScript switch

```
switch (expression)  {
   case constant_expression1 : statement1;
          ...
   case constant_expressionn : statementn;
   [default: statementn+1]
}
```

  - The control expression and the constant expressions some discrete type including integer types as well as character and enumeration types
  - The selectable statements can be statement sequences, blocks, or compound statements
  - Any number of segments can be executed in one execution of the construct (there is no implicit branch at the end of selectable segments)
  - `default` clause is for unrepresented values (if there is no `default`, the whole statement does nothing)
  - Any number of segments can be executed in one execution of the construct (a trade-off between reliability and flexibility—convenience.)
  - To avoid it, the programmer must supply a break statement for each segment.
- C# switch
  - C# switch statement differs from C-based in that C# has static semantic rule disallows the implicit execution of more than one segment
  - The rule is that every selectable segment must end with an explicit unconditional branch statement either a `break`, which transfers control out of the switch construct, or a `goto`, which can transfer control to on of the selectable segments. C# switch statement example:

```
switch (value)   {
   case -1:    Negatives++;
               break;
   case  0:    Zeros++;
               goto case 1;
   case  1:    Positives ++;
   default:    Console.WriteLine("Error in switch \n");
}
```

- Multiple Selection Using `if`
  - Multiple Selectors can appear as direct extensions to two-way selectors, using else-if clauses
  - Ex, Python selector statement (note that else-if is spelled `elif` in Python):

```
if count < 10 :
   bag1 = True
elif count < 100 :
   bag2 = True
elif count < 1000 :
   bag3 = True
```

    which is equivalent to the following:

```
if count < 10 :
   bag1 = True
else :
   if Count < 100 :
      bag2 = True
   else :
      if Count < 1000 :
         bag3 = True
```

    - The `elsif` version is the more readable of the two.

  - The Python example can be written as a Ruby `case`

```
case
   when count < 10        then bag1 = true
   when count < 100       then bag2 = true
   when count < 1000      then bag3 = true
end
```

  - Notice that this example is not easily simulated with a **switch-case** statement, because each selectable statement is chosen on the basis of a Boolean expression
  - In fact, none of the multiple selectors in contemporary languages are as **general** as the if-then-else-if statement

8.3 Iterative Statements 343

- An iterative statement is one that cause a statement or collection of statements to be executed zero, one, or more times
- The repeated execution of a statement or compound statement is accomplished either by iteration or recursion
- An iterative statement is often called **loop**
- Iteration is the very essence of the power of computer
- The repeated execution of a statement is often accomplished in a functional language by recursion rather than by iteration
- General design issues for iteration control statements:
  - How is iteration controlled?
  - Where should the control mechanism appear in the loop statement?
- The primary possibilities for iteration control are logical, counting, or a combination of the two
- The main choices for the location of the control mechanism are the top of the loop or the bottom of the loop
- The **body** of a loop is the collection of statements whose execution is controlled by the iteration statement
- The term **pretest** means that the loop completion occurs before the loop body is executed
- The term **posttest** means that the loop completion occurs after the loop body is executed
- The iteration statement and the associated loop body together form an **iteration statement**

8.3.1 Counter-Controlled Loops

- A counting iterative control statement has a variable, called the **loop variable**, in which the count value is maintained
- It also includes means of specifying the **intial** and **terminal** values of the loop variable, and the difference between sequential loop variable values, called the **stepsize**.
- The intial, terminal and stepsize are called the **loop parameters**.
- Design issues:
  - What are the type and scope of the loop variable?
  - 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?
  - Should the loop parameters be evaluated only once, or once for every iteration?
  -
- Fortran 90's **DO** syntax:

```
[name:]  DO label variable = initial, terminal [, stepsize]
           . . .
END DO [name]
```

  - The label is that of the last statement in the loop body, and the stepsize, when absent, defaults to **1**.
  - Loop variable **must** be an INTEGER and may be either negative or positive.
  - The loop parameters are allowed to be expressions and can have negative or positive values.
  - They are evaluated at the beginning of the execution of the DO statement, and the value is used to compute an iteration count, which then has the number of times the loop is to be executed.
  - The loop is controlled by the iteration count, not the loop param, so even if the params are changed in the loop, which is legal, those changes cannot affect loop control.
  - The iteration count is an internal var that is inaccessible to the user code.
  - The DO statement is a single-entry structure

- The for statement of the C-based languages

```
for ([expr_1] ; [expr_2] ; [expr_3])
   loop body
```

  - The loop body can be a single statement, a compound statement, or a null statement

```
for (count = 1; count <= 10; count++)
   . . .
```

  - All of the expressions of C's `for` are optional
  - If the second expression is absent, it is an **infinite** loop
  - If the first and third expressions are absent, no assumptions are made
  - The C-based languages for design choices are:
    - There are no explicit loop variable or loop parameters
    - All involved variables can be changed in the loop body
    - 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
  - C's for is more **flexible** than the counting loop statements of Fortran and Ada, because each of the expressions can comprise multiple statements, which in turn allow multiple loop variables that can be of any type
  - Consider the following for statement:

```
for (count1 = 0, count2 = 1.0;
    count1 <= 10 && count2 <= 100.0;
    sum = ++count1 + count2, count2 *= 2.5)
      ;
```

    The operational semantics description of this is:

```
count1 = 0
count2 = 1.0
loop:
   if count1 > 10 goto out
   if count2 > 100.0 goto out
   count1 = count1 + 1
   sum = count1 + count2
   count2 = count2 * 2.5
   goto loop
out:
   . . .
```

    - The loop above does not need and thus does **not** have a loop body
  - The for statement of C99 and C++ differs from earlier version of C in two ways:
    - It can use an arithmetic expression or a Boolean expression for loop control
    - The first expression can include variable definitions (scope is from the definition to the end of the loop body), for example

```
for (int count = 0; count <= 10; count++) { . . . }
```

  - The for statement of Java and C# is like that of C++, except that the loop control expression is restricted to Boolean

- The for statement of Python
  - The general form of Python's `for` is:

```
for loop_variable in object:
    - loop body
[else:
    - else clause]
```

  - The `object` is often 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
  - Consider the following example:

```
for count in [2, 4, 6] :
    print count
```

    produces

```
2
4
6
```

8.3.2 Logically Controlled Loops

- Repetition control is based on a Boolean expression rather than a counter
- Design Issues:
  - Should the control be pretest or posttest?
  - Should the logically controlled loop be a special form of a counting loop or a separate statement?
- The C-based programming languages include both pretest and posttest logically controlled loops that are not special forms of their counter-controlled iterative statements
- The **pretest** and **posttest** logical loops have the following forms **(**`while` and `do-while`**)**:

```
while (control_expression)
   loop body
```

and

```
do
   loop body
while (control_expression);
```

- These two statements forms are exemplified by the following C# code:

```
sum = 0;
indat = Int32.Parse(Console.ReadLine( ));
while (indat >= 0) {
   sum += indat;
   indat = Int32.Parse(Console.ReadLine( ));
}

value = Int32.Parse(Console.ReadLine( ));
do {
   value /= 10;
   digits ++;
} while (value > 0);
```

- The only real difference between the do and the while is that the do always causes the loop body to be executed **at least once**
- Java's `while` and `do` statements are similar to those of C and C++, except the control expression must be Boolean type, and because Java does **not** have a goto, the loop bodies cannot be entered anywhere but at their beginning

8.3.3 User-Located Loop Control Mechanisms

- It is sometimes convenient for a programmer to choose a location for loop control other than the top or bottom of the loop
- Design issues:
  - Should the conditional mechanism be an integral part of the exit?
  - Should only one control body be exited, or can enclosing loops also be exited?
- C and C++ have unconditional unlabeled exits (`break`)
- Java, Perl, and C# have unconditional labeled **exits** (`break` in Java and C#, `last` in Perl)
- The following is an example of nested loops in C#:

```
OuterLoop:
   for (row = 0; row < numRows; row++)
      for (col = 0; col < numCols; col++) {
         sum += mat[row][col];
         if (sum > 1000.0)
              break outerLoop;
   }
```

- C and C++ include an unlabeled control statement, `continue`, that transfers control to the control mechanism of the smallest enclosing loop
- This is not an exit but rather a way to **skip** the rest of the loop statements on the current iteration without terminating the loop structure. Ex:

```
while (sum < 1000) {
   getnext(value);
   if (value < 0) continue;
   sum += value;
}
```

- A negative value causes the assignment statement to be **skipped**, and control is transferred instead to the conditional at the top of the loop
- On the other hand, in

```
while (sum < 1000) {
   getnext(value);
   if (value < 0) break;
   sum += value;
}
```

  - A negative value **terminates** the loop

- Java, Perl, and C# have statements similar to continue, except they can include labels that specify which loop is to be continued
- The motivation for user-located loop exits is simple: They fulfill a common need for goto statements through a highly restricted branch statement
- The target of a goto can be many places in the program, both above and below the goto itself
- However, the targets of user-located loop exits must be below the exit and can only follow immediately the end of a compound statement

8.3.4 Iteration Based on Data Structures

- A general data-based iteration statement uses a user-defined data structure and a user-defined function (the **iterator**) to go through the structure's elements
- The iterator is called at the beginning of each iteration, and each time it is called, the iterator return a n element from a particular data structure in some specific order
- C's `for` can be used to build a user-defined iterator:

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

- Java 5.0 uses `for`, although it is called `foreach`
  - The following statement would iterate though all of its elements, setting each to myElement:

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

  - The new statement is referred to as "`foreach`," although is reserved word is `for`

- C#'s `foreach` statement iterates on the elements of array and other collections
  - C# and F# (and the other .NET languages) have generic library classes, like Java 5.0 (for arrays, lists, stacks, and queues)
  - For example, there are **generic** collection classes for lists, which are dynamic length array, stacks, queues, and dictionaries (has table)
  - All of these predefined generic collections have built-in iterator that are used implicitly with the `foreach` statement
  - Furthermore, users can define their own collections and write their own iterators, implement the `IEnumerator` interface, which enables the use of use `foreach` on these collections

```
List<String> names = new List<String>();
names.Add("Bob");
names.Add("Carol");
names.Add("Ted");
foreach (Strings name in names)
      Console.WriteLine (name);
```

8.4 Unconditional Branching 355

- An unconditional branch statement transfers execution control to a specified place in the program
- The unconditional branch, or **goto**, is the most powerful statement for controlling the flow of execution of a program's statements
- However, using the goto carelessly can lead to **serious problems**
- Without restrictions on use, imposed by either language design or programming standards, goto statements can make programs very difficult to read, and as a result, highly **unreliable** and costly to **maintain**
- There problems follow directly from a goto's ability to force any program statement to follow any other in execution sequence, regardless of whether the statement proceeds or follows previously executed statement in textual order
- Java, Python, and Ruby do **not** have a goto. However, most currently popular languages include a goto statement
- **C#** uses goto in the **switch** statement

8.5 Guarded Commands 356

- New and quite different forms of selection and loop structures were suggested by Dijkstra (1975)
- His primary motivation was to provide control statements that would support a **new** program design methodology that ensured correctness (verification) during development rather than when verifying or testing completed programs
- Basis for two linguistic mechanisms for concurrent programming in CSP (Hoare, 1978)
- Basic idea: if the order of evaluation is not important, the program should **not** specify one
- Dijkstra's selection guarded command has the 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
- Ex

```
if i = 0 -> sum := sum + i
[] i > j -> sum := sum + j
[] j > i -> sum := sum + i
fi
```

  - If `i = 0` and `j > i`, this statement chooses non-deterministically between the first and third assignment statements
  - If `i` is equal to `j` and is not zero, a run-time error occurs because none of the condition are true

- Ex

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

  - This computes the desired result without over specifying the solution
  - In particular, if x and y are equal, it does **not** matter which we assign to max
  - This is a form of abstraction provide by the non-deterministic semantics of the statement

- The loop structure proposed by Dijkstra has the 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

- Ex Consider the following problem: Given four integer variables, q1, q2, q3, and q4, rearrange the values of the four so that $q1 \leq q2 \leq q3 \leq q4$.
  - **Without** guarded commands, one straightforward solution is to put the four values into an array, sort the array, and then assign the values from the array back into the scalar variables q1, q2, q3, and q4. While this solution is not difficult, it requires a good deal of code, especially if the sort process must be included.
  - Now, uses guarded commands to solve the same problem but in a more concise and **elegant** way

```
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
```

- Dijkstra's guarded command control statements are **interesting**, in part because they illustrate how the syntax and semantics of statements can have an impact on program verification and vice versa.
- Program verification is impossible when **goto** statements are used
- Verification is greatly simplified if
  - only selection and logical pretest loops
  - only guarded commands

Summary 360

- Control statements occur in several categories:
  - Selection Statements
  - Iterative Statements
  - Unconditional branching
- The **switch** statement of the C-based languages is representative of multiple-selection statements
- C's **for** statement is the most flexible iteration statement although its flexibility lead to some reliability problem
- Data-based iterators are loop statements for processing data structures, such as linked lists, hashes, and trees.
  - The **for** statement of the C-based languages allows the user to create iterators for user-defined data
  - The **foreach** statement of Perl and C# is a predefined iterator for standard data structure
- The unconditional branch, or **goto**, is the most powerful statement for controlling the flow of execution of a program's statements
  - The unconditional branch, or goto, has been part of **most** imperative languages
  - Its problems have been widely discussed and debated.
  - The current consensus is that it should remain in most languages but that its **dangers** should be **minimized** through programming discipline
- Dijkstra's **guarded commands** are alternative control statement with positive theoretical characteristics.
  - Although they have not been adopted as the control statements of a language, part of the semantics appear in the concurrency mechanisms of CSP and the function definitions of Haskell