Chapter 7

Statement-Level Control Structures

Chapter 8 Topics

- Introduction
- Selection Statements
- Iterative Statements
- Unconditional Branching
- Guarded Commands
- Conclusions

LEVELS OF CONTROL FLOW

- Within expressions
  - low level, discussed in chapter 7
- Among program units
  - high level, will be discussed in chapter 9
- Among program statements

CONTROL STATEMENTS: EVOLUTION

- FORTRAN I control statements were based directly on IBM 704 hardware
- Much research and argument in the 1960s about the issue
  - One important result:

    It was proven that all algorithms represented by flowcharts can be coded with only two-way selection and pretest logical loops

CONTROL STRUCTURE

- Control statements
  - Too many – less readable
  - Too few – require the use of low level statement, which is also less readable
- A *control structure* is a control statement and the statements whose execution it controls
- Design question
  - Should a control structure have multiple entries?

Selection Statements

- A *selection statement* provides the means of choosing between two or more paths of execution
- Two general categories:
  - Two-way selectors
  - Multiple-way selectors

Two-Way Selection Statements

- General form:

```
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?

Two-Way Selection: Examples

- FORTRAN: **`IF`** (boolean_expr) statement
- Problem: can select only a single statement; to select more, a **`GOTO`** must be used, as in the following example

```
IF (.NOT. condition) GOTO 20
      ...
      20 CONTINUE
```

- Negative logic is bad for readability
- This problem was solved in FORTRAN 77
- Most later languages allow compounds for the selectable segment of their single-way selectors

TWO-WAY SELECTION: EXAMPLES

- ALGOL 60:

```
if (boolean_expr)
    then statement (then clause)
    else statement (else clause)
```

- The statements could be single or compound

NESTING SELECTORS

- Java example

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

- Which `if` gets the `else`?
- Java's static semantics rule: `else` matches with the nearest `if`
- Indentation has no effect on semantics

Nesting Selectors (continued)

- To force an alternative semantics, compound statements may be used:

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

- The above solution is used in C, C++, and C#
- Perl requires that all then and else clauses to be compound

Multiple-Way Selection Statements

- Allow the selection of one of any number of statements or statement groups
- Design Issues:
  1. What is the form and type of the control expression?
  2. How are the selectable segments specified?
  3. Is execution flow through the structure restricted to include just a single selectable segment?
  4. What is done about unrepresented expression values?

Multiple-Way Selection: Examples

- Early multiple selectors:
  - FORTRAN arithmetic `IF` (a three-way selector)

```
IF (arithmetic expression) N1, N2, N3
```

  - Segments require `GOTO`s
  - Not encapsulated (selectable segments could be anywhere)

Multiple-Way Selection: Examples

- Modern multiple selectors
  - C's `switch` statement

```
switch (expression) {
      case const_expr_1: stmt_1;
      …
      case const_expr_n: stmt_n;
      [default: stmt_n+1]
}
```

Multiple-Way Selection: Examples

- Design choices for C's **`switch`** statement
  1. Control expression can be only an integer type
  2. Selectable segments can be statement sequences, blocks, or compound statements
  3. Any number of segments can be executed in one execution of the construct (there is no implicit branch at the end of selectable segments)
  4. **`default`** clause is for unrepresented values (if there is no **`default`**, the whole statement does nothing)

Multiple-Way Selection: Examples

- The Ada `case` statement

```
case expression is
    when choice list => stmt_sequence;
    …
    when choice list => stmt_sequence;
    when others => stmt_sequence;]
end case;
```

- More reliable than C's `switch` (once a stmt_sequence execution is completed, control is passed to the first statement after the `case` statement

Multiple-Way Selection Using `if`

- Multiple Selectors can appear as direct extensions to two-way selectors, using else-if clauses, for example in Ada:

```
if ...
  then ...
elsif ...
  then ...
elsif ...
  then ...
  else ...
end if
```

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. What is the value of the loop variable at loop termination?
  3. 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?
  4. Should the loop parameters be evaluated only once, or once for every iteration?

Iterative Statements: Examples

- FORTRAN 90 syntax

```
DO label var = start, finish [, stepsize]
```

- Stepsize can be any value but zero
- Parameters can be expressions
- Design choices:
  1. Loop variable must be **INTEGER**
  2. Loop variable always has its last value
  3. 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
  4. Loop parameters are evaluated only once

Iterative Statements: Examples

- FORTRAN 95 : a second form:

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

  - Loop variable must be an **`INTEGER`**

Iterative Statements

- Pascal's `for` statement

```
for variable := initial (to|downto) final do
   statement
```

- Design choices:
  1. Loop variable must be an ordinal type of usual scope
  2. After normal termination, loop variable is undefined
  3. The loop variable cannot be changed in the loop; the loop parameters can be changed, but they are evaluated just once, so it does not affect loop control

Iterative Statements: Examples

- Ada

```
for var in [reverse] discrete_range loop
...
end loop
```

- A discrete range is a sub-range of an integer or enumeration type
- Scope of the loop variable is the range of the loop
- Loop variable is implicitly undeclared after loop termination

ITERATIVE STATEMENTS: EXAMPLES

- C's `for` statement

```
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
- 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

ITERATIVE STATEMENTS: 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

Iterative Statements: Logically-Controlled Loops

- Repetition control is based on a Boolean
- Design issues:
  - Pre-test or post-test?
  - Should the logically controlled loop be a special case of the counting loop statement ? expression rather than a counter
- General forms:

```
while (ctrl_expr)   do
   loop body              loop body
                    while (ctrl_expr)
```

Iterative Statements: Logically-Controlled Loops: Examples

- Pascal has separate pre-test and post-test logical loop statements (`while-do` and `repeat-until`)
- C and C++ also have both, but the control expression for the post-test version is treated just like in the pre-test case (`while-do` and `do- while`)
- Java is like C, except the control expression must be Boolean (and the body can only be entered at the beginning -- Java has no **`goto`**

Iterative Statements: Logically-Controlled Loops: Examples

- Ada has a pretest version, but no post-test
- FORTRAN 77 and 90 have neither
- Perl has two pre-test logical loops, `while` and `until`, but no post-test logical loop

Iterative Statements: 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?

Iterative Statements: User-Located Loop Control Mechanisms break and continue

- C , C++, and Java: **`break`** statement
- Unconditional; for any loop or **`switch`**; one level only
- Java and C# have a labeled `break` statement: control transfers to the label
- An alternative: **`continue`** statement; it skips the remainder of this iteration, but does not exit the loop

```
for (a = 0; a < 5; a++)
{
if (a == 2) break;
printf("a = %d\n", a);
}
```

```
for (a = 0; a < 5; a++)
{
if (a == 2) continue;
printf("a = %d\n", a);
}
```

Iterative Statements: 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)){
}
```

Iterative Statements: Iteration Based on Data Structures (continued)

- 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
- Well-known mechanism: `goto` statement
- Major concern: Readability
- Some languages do not support `goto` statement (e.g., Module-2 and Java)
- C# offers `goto` statement (can be used in `switch` statements)
- Loop exit statements are restricted and somewhat camouflaged `goto`'s

GUARDED COMMANDS

- Suggested by Dijkstra
- Purpose: to support a new programming methodology that supports 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

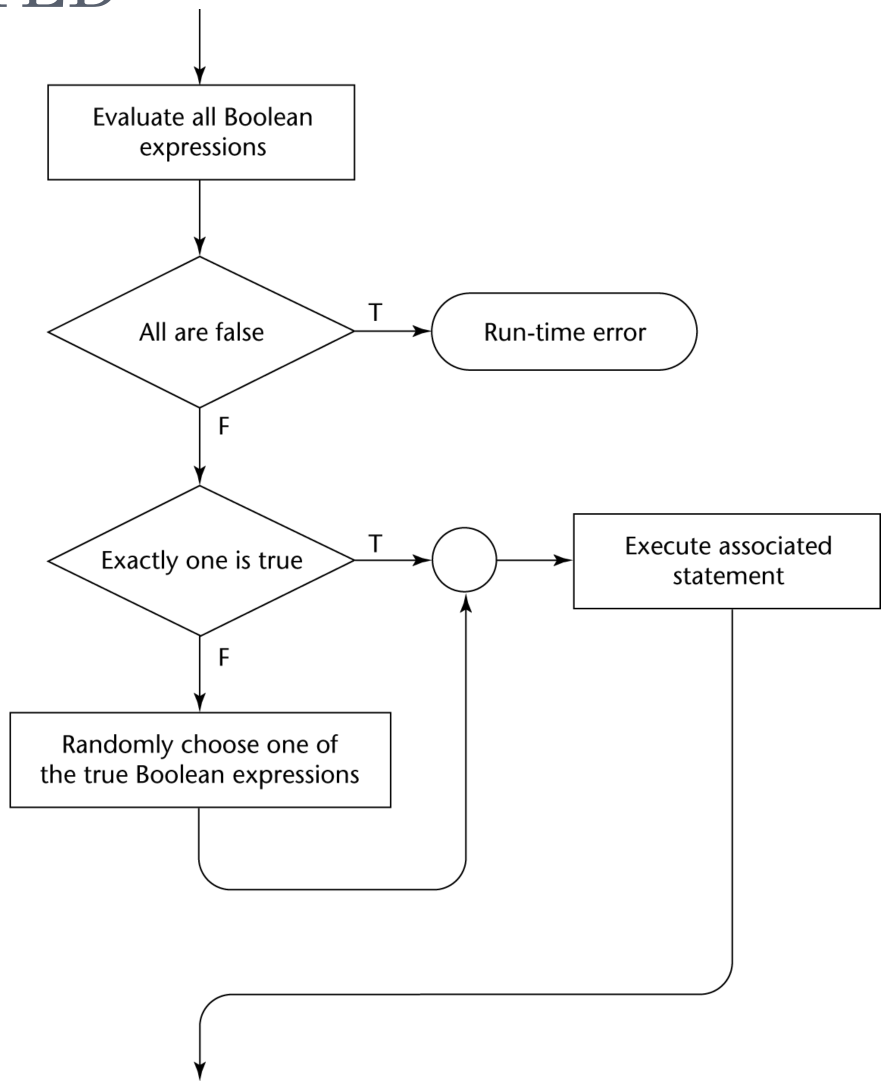
SELECTION GUARDED COMMAND

- 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

Selection Guarded Command: Illustrated



Selection Guarded Command Example

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

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

- How would you do with ordinary coding?

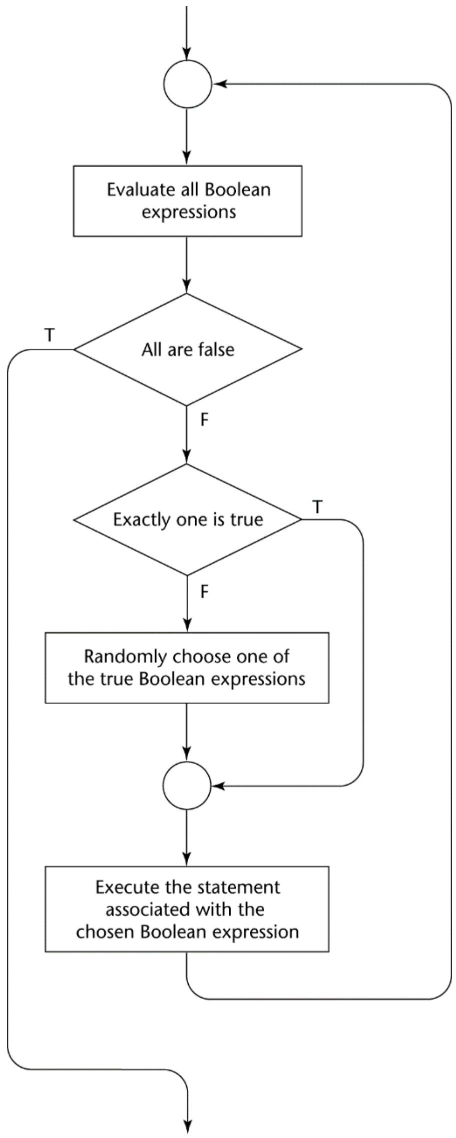
Loop Guarded Command

- **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: Illustrated



LOOP GUARDED COMMAND EXAMPLE

```
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
```

- How would you do with ordinary coding?

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

Conclusion

- 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