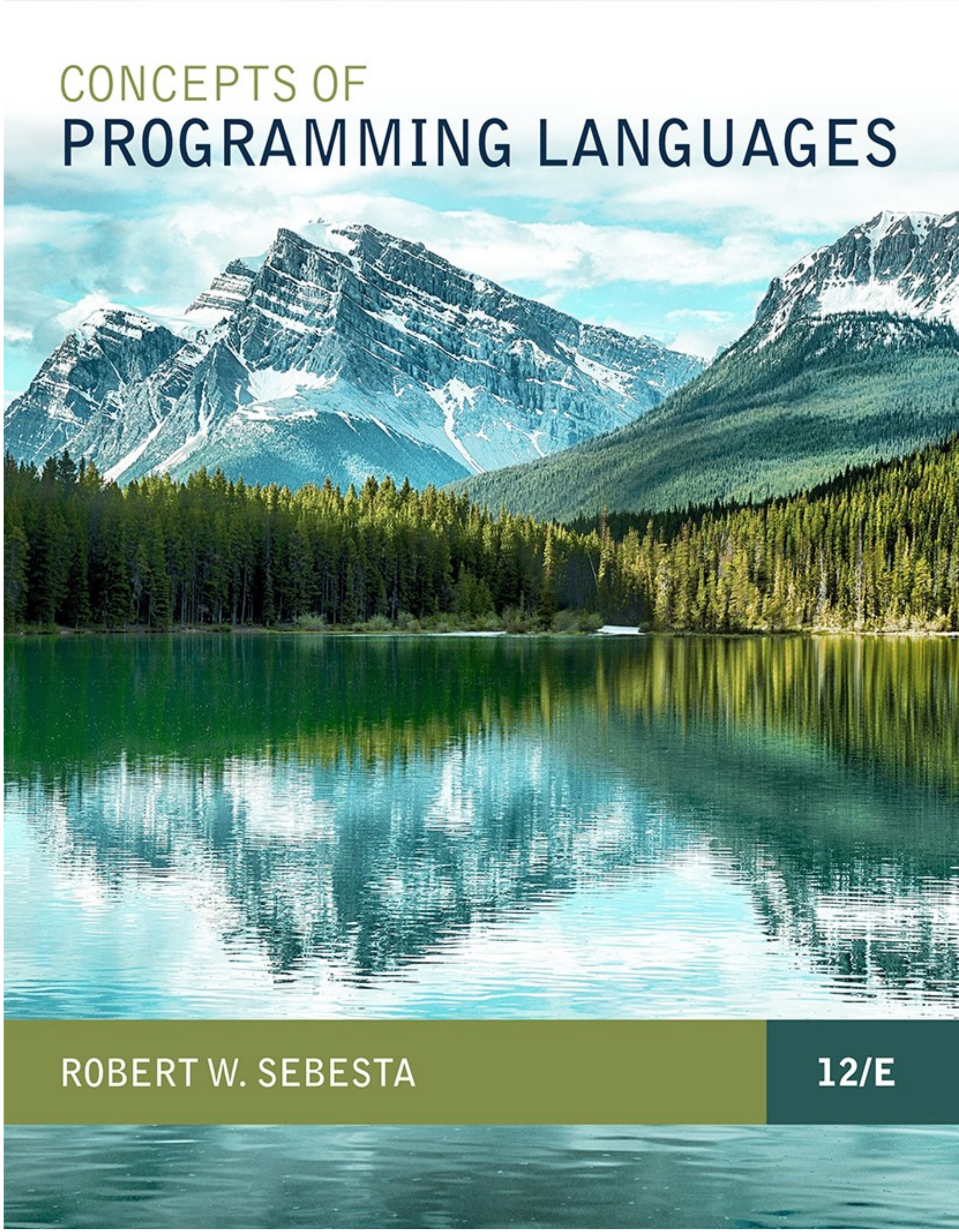
Chapter 8

Statement-Level Control Structures



ISBN 0-321-49362-1

Chapter 8 Topics

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

Selection Statements

- Selection statement :
  - provides the means of choosing between two or more paths of execution
- Two general categories:
  - Two-way selectors (e.g., **if** … **else** ...)
  - Multiple-way selectors (e.g., **switch** … **case**...)

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?

Control Expression

- If the *then* reserved word or some other syntactic marker is not used to introduce the **then clause**
  - the control expression is placed in parentheses
  - E.g., Java: **if** (*boolean-expression*) {…} **else** {…}
- In C89, C99, Python, and C++
  - the control expression can be arithmetic
  - E.g., **if** (a-b) … **else** ...
- In most other languages
  - the control expression **must** be Boolean

Clause Form

- The *then* and *else clauses*
  - can be **single** statements or **compound** statements
- In Perl, all clauses must be delimited by braces

E.g.,

```
if ($age >= 18) {
        print "you can vote.\n"; }
else {
        print "You are too young to vote\n"; }
```

- Python uses indentation to define clauses

E.g.,

```
var = 100
if var:
        print "Got a positive value"
        print “Yes”
else:
        print "Got a negative value"
```

Nesting Selectors : Java

- 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 previous `if`

Nesting Selectors: Java

- 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#

Nesting Selectors: Ruby

- Statement sequences as clauses: Ruby

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

Nesting Selectors: Python

- Python: uses indentation

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

Selector Expressions

- In ML, F#, and Lisp
  - the selector is an expression
- in F#:

```
let y =
   if x > 0 then x
   else 2 * x
```

  - The types of the values returned by *then* and *else* clauses must be the same.

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. How are case values specified?
  5. What is done about unrepresented expression values?

Multiple-Way Selection

- C, C++, Java, and JavaScript

```
switch (expression) {
   case const_expr1: stmt1;
   …
   case const_exprn: stmtn;
   [default: stmtn+1]
}
```

Multiple-Way Selection: C

- Design choices for C's **`switch`** statement
  - Control expression can be only an integer type
  - Selectable segments could be:
    - Statement sequences (e.g., a = b + c;);
    - Blocks (e.g., { …. });
    - Compound statements (e.g., while() {})
  - Any number of segments can be executed in one execution of the construct
  - **`default`** clause is for unrepresented values (if there is no **`default`**, the whole statement does nothing)
  - E.g. ,

```
switch(input) {
case 1: printf("case #1\n"); break;
case 2: printf("case #2\n"); break
default: printf( "default case\n" ); break; }
```

Multiple-Way Selection: C#

- C#
  - Differs from C in that it has a static semantics rule that **disallows** the implicit execution of more than one segment
  - Each selectable segment **must** end with an unconditional branch (**`goto`** or **`break`**)
  - the control expression and case constants can be **strings**
  - E.g. ,

```
int switchCase=1;
switch (switchCase) {
case1: Console.WriteLine("Case 1"); break;
case2: Console.WriteLine("Case 2"); break;
default: Console.WriteLine("Default case"); break;
}
```

Multiple-Way Selection: Ruby

- Ruby case statements:

```
puts case input
when 1..10
        "It's between 1 and 10"
when 100
        "It's 100"
when String
        "You passed a string"
else
        " I have no idea what to do with that."
end
```

Multiple-Way Selection: Python

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

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

Multiple-Way Selection: Ruby

- 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
```

Multiple-Way Selection: Scheme

- General form of a call to **COND**:

```
(COND
  (predicate₁    expression₁)
  …
  (predicateₙ    expressionₙ)
  [(ELSE         expressionₙ₊₁)]
)
```

  - The ELSE clause is optional; ELSE is a synonym for true
  - Each predicate-expression pair is a parameter
  - Semantics: The value of the evaluation of COND is the **value** of the expression associated with the first predicate expression that is true

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
  - a means of specifying the *initial*, *terminal*
    , and *stepsize* values
- Design Issues:
  1. What are the type, 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?

Counter-Controlled Loops: C

- C-based languages

  `for ([init_expr];[test_expr];[update_expr]) statement`

  - The **expressions** can be statements, or 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 2nd expression, **test_expr**, is absent, it is an infinite loop
    - E.g., for (p=0; p+=(a&1)*b, a!=1; a++,b++) { …. }
    - for (p=0; ; a++,b++) { …. }
- 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

Counter-Controlled Loops: C++

- 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)

     E.g., for (int I = 0; I < 100; i++) { …}
- Java and C#
  - Differs from C++ in that the control expression must be **Boolean**

Counter-Controlled Loops: Python

- 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 a range function, e.g., `range(5)` 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 only if the loop exhausted iterating the list.
  - E.g., ...

```
for n in range(10,15):
    for x in range(2, n):
        if n % x == 0:
            print '%d = %d * %d' % (n, x, n/x)
            break
    else:
        print n, ' is a prime number'
```

10 = 2 * 5
11 is a prime number
12 = 2 * 6
13 is a prime number
14 = 2 * 7

Counter-Controlled Loops: F#

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

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

  - recursive function **forLoop** with the parameters:
    - ***loopBody***--function defines the loop's body
    - ***reps***: 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 **special** case of:
    - counting loop statement OR
    - separate statement?

Logically-Controlled Loops: C/C++

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

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

Logically-Controlled Loops: F#

- 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`.

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

- Java, C , C++, Python, Ruby, C#:
  - have unconditional unlabeled exits--`break`
- Perl
  - have unconditional labeled exits--`last`
- C, C++, Python
  - an unlabeled control statement, `continue`, skips the remainder of the current iteration, but does not exit the loop
- Java, Perl
  - have labeled versions of `continue`
  - `E.g., Java,  continue test; // test is a label`

Iteration Based on Data Structures

- The number of elements in a data structure controls loop iteration
- *Iterator:*
  - 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

- 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

- C# and F#:
  - Have generic library classes like Java 5.0
    - E.g., 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

- 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!"}    # run 3 times
list.each {|value| puts value}
```

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

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

Iteration Based on Data Structures

- Ruby blocks
  - have parameters (in vertical bars)
  - are executed when the method executes a **`yield`** function

```
def fibonacci(last)
  first, second = 1, 1
  while first <= last
    yield first
    first, second = second, first + second
  end
 end
puts "Fibonacci numbers less than 100 are:"
fibonacci(100) {|num| print num, " "}   # block
puts
## output
Fibonacci numbers less than 100 are:
1 1 2 3 5 8 13 21 34 55 89
```

Unconditional Branching

- Transfers execution control to a *specified place* in the program
- Major concern: Readability
- Some languages do **not** support `goto` statement (e.g., Java)
- C# offers `goto` statement (can be used in **`switch`** statements)

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

- E. W. Dijkstra's Form:

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

, where Boolean expr: a guard

statement (statement sequences): guarded command

[] : fatbar used to separate selection statement

- Semantics: when the if-block is reached,
  - Evaluate **all** Boolean expressions
  - If more than one are true, choose one **non-deterministically**, e.g., the **first** one Or the **third** one
  - If none is true, it is a runtime error
  - Force programmer to consider and list all possibilities
  - E.g.,

```
if  I = 0  ->  sum := sum + I
    [] I > J   ->  sum := sum + J
    [] J > I   ->  sum := sum + I
fi
```

Loop Guarded Command

- E. W. Dijkstra's **Form**

```
do <Boolean expr> -> <statement>
[] <Boolean expr> -> <statement>
 ...
[]  <Boolean expr> -> <statement>
od
```

- Semantics: for each iteration
  - Evaluate **all** Boolean expressions on each iteration
  - If more than one are true, choose one **non-deterministically**
  - If none is true, exit loop
  - E.g.,
```
do   q1 > q2 -> t := q1; q1 := q2; q2 := t;
  []   q2 > q3 -> t := q2; q2 := q3; q3 := t;
  []   q3 > q4 -> t := q3; q3 := q4; q4 := t;
od
```

E.g., Boolean_expr: check service health from one data center