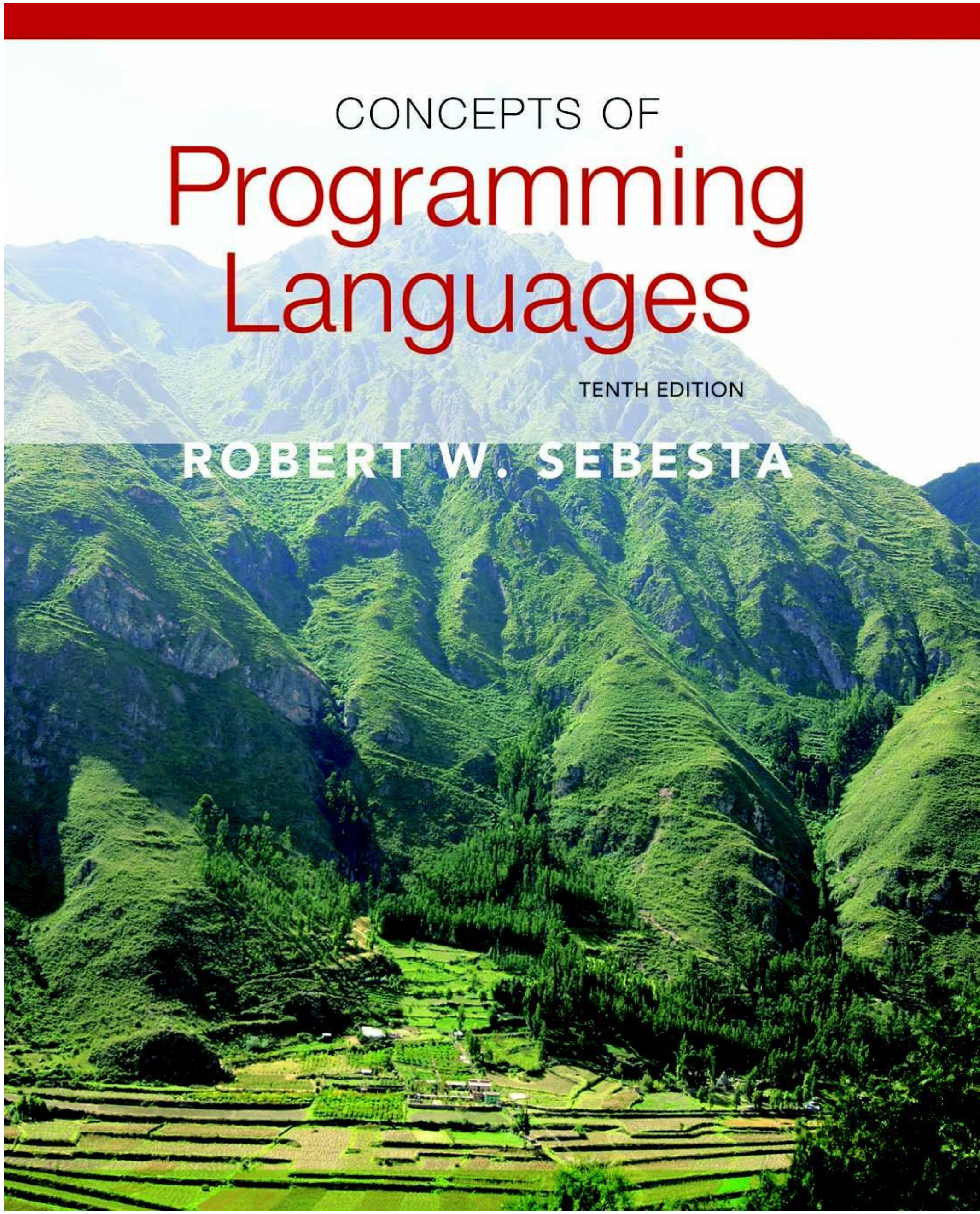
Chapter 8

Statement-Level Control Structures



Chapter 8 Topics

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

Levels of Control Flow

- The flow of control within expressions (Chapter 7) is governed
  - By operator associativity
  - By precedence rules
- The flow of control among program units (Chapter 9 and 13)
- The flow of control among program statements (this chapter)

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 logically controlled iterations

Control Structure

- A **control structure** is a control statement and the collection of statements whose execution **it** (a control statement) controls

- Only one design issue:
  – Should a control structure have multiple entries?

제어구조가 멀티플 엔트리가 있으면 안되는가?

가질 수 있다
if else if else if else

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?

The Control Expression if문 안에 들어있는 판정문

- 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

- In C89, C99, Python, and C++, the control expression can be arithmetic

- In most other languages, the control expression must be Boolean

Clause Form

- In many contemporary languages, the then and else clauses **can be single statements or compound statements**
- In Perl, all clauses must be delimited by braces (they must be compound)
- In Fortran 95, Ada, Python, and Ruby, clauses are statement sequences

보통 if else는 가까운 것 끼리 match가 되지만
파이썬의 경우 인덴트가 같은것끼리 match가 된다

- **Python** uses **indentation** to **define clauses**

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


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 previous `if`

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#

Nesting Selectors (continued)

- Statement sequences as clauses: Ruby

```
if a > b then
    sum = sum + a
    acount = acount +1
else
    sum = sum + b
    bcount = bcount +1
end
```

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

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

Nesting Selectors (continued)

- Python

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

- Ruby

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

Semantically equivalent to

들여쓰기 잘못하면 완전히 다른 문장이 된다

Semantically not equivalent to

- Python

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

Selector Expressions (skip~)

- In ML, F#, and LISP, the selector is an expression
- F#

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

  - If the `if` expression returns a value, there must be an else clause (the expression could produce output, rather than a value)

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 expression that control the selection?
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 the case values specified?
5. How should unrepresented expression values be handled, if at all? (default……?)

Multiple-Way Selection: Examples

- C, C++, Java, and JavaScript

```
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 (p. 376)
  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) – (P.375~6 example)
     (`break` → explicit branch to separate segments logically)
  4. `default` clause is for unrepresented values (if there is no `default`, the whole statement does nothing)

묵시적으로 안해주니 반드시 써야함
ex) break

Multiple-Way Selection: Examples

바이패스를 많이 쓰는것은 좋지 않다.

- The C# switch statement differs from C in that it has a static semantics rule that disallows the implicit execution of more than one segment (p. 377)
  - Each selectable segment must end with **an unconditional branch** (`goto` or `break`)
  - Also, in C# the **control expression** and the **case constants** can be **strings**

Multiple-Way Selection: Examples (skip~)

- Ruby has two forms of case statements–we'll cover only one

```
leap = case
    when year % 400 == 0 then true
    when year % 100 == 0 then false
    else year % 4 == 0
    end
```

switch case의 경우 조건식을 사용하지 않는다고 했지만
언어에 따라 이렇게 사용하는 경우도 있다 정도만 기억

Implementing Multiple Selection Structures

- Approaches: *(p. 378~379 , C switch statement)*
  - Multiple conditional branches
  - Store case values in a table and use a linear search of the table
    - When there are more than ten cases, a hash table of case values can be used
    - If the number of cases is small and more than half of the whole range of case values are represented, an array whose indices are the case values and whose values are the case labels can be used

Multiple-Way Selection Using `if`

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

```
if count < 10 :
  bag1 = True
elif count < 100 :
  bag2 = True
elif count < 1000 :
  bag3 = True
else :
  bag4 = True
```

More readable
←

More typing
→

```
if count < 10 :
  bag1 = True
else :
  if count < 100 :
    bag2 = True
  else :
    if count < 1000 :
      bag3 = True
    else :
      bag4 = True
```

What is the difference between switch and else-if ?

※ Else-if (multiple selection using if)
When selections must be made on the basis of a Boolean expression rather than some ordinal type

When selections must be made on the basis of a boolean expression rather than some ordinal type

Multiple-Way Selection Using `if`

- 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
```

Scheme's Multiple Selector (skip~)

- 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

모든 for는 while로 변환가능
역방향은 X

- General design issues for iteration control statements:
  1. How is the iteration controlled?
  2. Where should the control mechanism appear in the loop statement? (pretest? or posttest?)

Counter-Controlled Loops

- A <u>**counting iterative statement**</u> 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?

증감(i + +, i—)

※ **The initial, terminal, and stepsize specifications of a loop are called the loop parameters.**

Counter-Controlled Loops: Examples (Ada) skip~

- Ada

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

- Design choices:
  - Type of the loop variable is that of the discrete range (A discrete range is a sub-range of an integer or enumeration type).
  - Loop variable does not exist outside the loop
  - The loop variable cannot be changed in the loop, but the discrete range can; it does not affect loop control
  - The discrete range is evaluated just once
  - Cannot branch into the loop body

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

An example of a skeletal C for statement (P. 385)

```
for (count =1; count <= 10; count++) {
      …
/* loop body - a single statement or compound statement
*/
}
```

/* 1st expression : initialization, evaluated only once.

2nd expression : loop control (Boolean type - 0:false or 1:true),
evaluated before each execution of the loop body.
If the second expression is absent, it is an infinite loop.
(The reason is that an absent 2nd expression is considered true)

3rd expression : executed after each execution of the loop body */

C `for` statement (P. 386) – no loop body

```
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:…
```

Counter-Controlled Loops: Examples

- C99 and C++ differs from earlier versions of C in two ways:
  1. The control expression can also be **Boolean** (addition to an **arithmetic expression**)
  2. The initial expression can include variable definitions **(The scope is from the definition to the end of the loop body)**

```
for (int count = 0, count < len; count++) { … }
```

- Java and C#
  - Differs from C++ in that the **control expression** must be **Boolean**

Counter-Controlled Loops: Examples (p. 387)

교과서 395 참고

- Python

  `for` loop_variable `in` object`:`

   - loop body

  [`else`:

   - else clause]

range(5) [0,1,2,3,4]리턴
range(2,7) [2,3,4,5,6]리턴
range(0,8,2) [0,2,4,6]리턴

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

produces

2
4
6
```

– 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` (p. 387)

– 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

Counter-Controlled Loops: Examples (skip~)

- 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 rather than a counter
- Actually, logically controlled loops are more general than counter-controlled loops
- 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?

Logically-Controlled Loops: Examples

- 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)
```

  - The operational semantics descriptions (p. 389)
  - 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 (skip~)

- 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 (P.390~391)

- Sometimes it is convenient for **the programmers** to **decide a location** for **loop control** (<u>other</u> than <u>top or bottom of the loop</u>)
- Simple design for single loops (e.g., `break`)
- Design issues for nested loops
  1. Should the conditional mechanism be an integral part of the exit?
  2. Should only one loop body be exited, or can enclosing loops also be exited?

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`

Continue, Break vs goto

```
/*
  Java continue with label example.
*/
import java.io.*;
class ContinueWithLabelDemo {
    public static void main(String[] args) {
        ...
    test:
        for (int i = 0; i <= max; i++) {
                ...
            while (n-- != 0) {
                if (searchMe.charAt(j++)
                        != substring.charAt(k++)) {
                    continue test;
                }
            }
            foundIt = true;
                break test;
        }
        System.out.println(foundIt ? "Found it" : "Didn't find it");
    }
}
```

※ goto: 무조건 지정한 라벨 위치로 분기 (위치와 무관)
※ continue, break: 블록 시작위치 또는 바깥 위치로 분기
(라벨명의 위치에 따라 결정됨)

Iteration Based on Data Structures

- The number of elements in a data structure controls loop iteration (p. 391~2 examples)
- 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; ptr = traverse(ptr)){
   ...
} /* traverse is a function that sets its parameter (ptr) to point
     to the next element of a tree in the desired order */
```

Iteration Based on Data Structures (continued)

- PHP : predefined **iterators (p.393 example)**
  - `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 (enhanced version of the for)
  (reserved word is `for`, although it is called **foreach**)

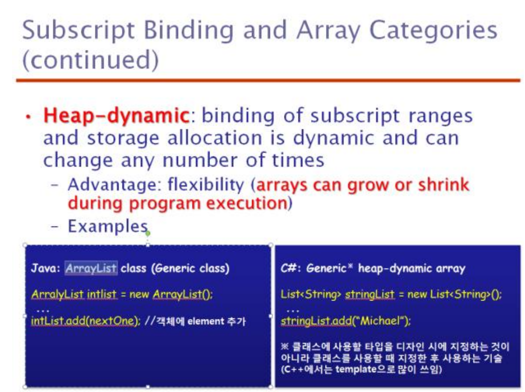
For arrays and any other class that implements the `Iterable` interface, e.g., `ArrayList` collection

```
String myList ={"a", "b", "c", "d"}
for (String myElement : myList) {
   System.out.println(myElement)
}
```

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**.

(C# example)




```
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);
```

리스트에 있는내용 출력

Iteration Based on Data Structures (continued)

- In Ruby, a *block* is a sequence 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`**): (p.394~5 example)

파라미터(매개변수)

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

for x in 1…5
print x

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

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

- Ruby has a `for` statement, but Ruby converts them to `upto` method calls

Iteration Based on Data Structures (continued)

- Ada
  - Ada allows the range of a loop iterator and the subscript range of an array be connected

```
subtype MyRange is Integer range 0.99;
MyArray: array (MyRange) of Integer;
for Index in MyRange loop
  ...MyArray(Index) ...
end loop;
```

Unconditional Branching

goto

책 384
goto case 1: 과 같은 형태로
제한적인 형태로 사용하는 경우가 존재한다.

- 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) – discussed in section 8.2.2.2.
- Loop exit statements are **restricted** and **somewhat camouflaged**(위장된/일종의) **goto statements**.

Guarded Commands

- Designed by Dijkstra
- Purpose: to support a new programming methodology that ensured **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 (skip~)

- 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

Loop Guarded Command (skip~)

- 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 (skip~)

- 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

Conclusions

- 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 use quite different control structures