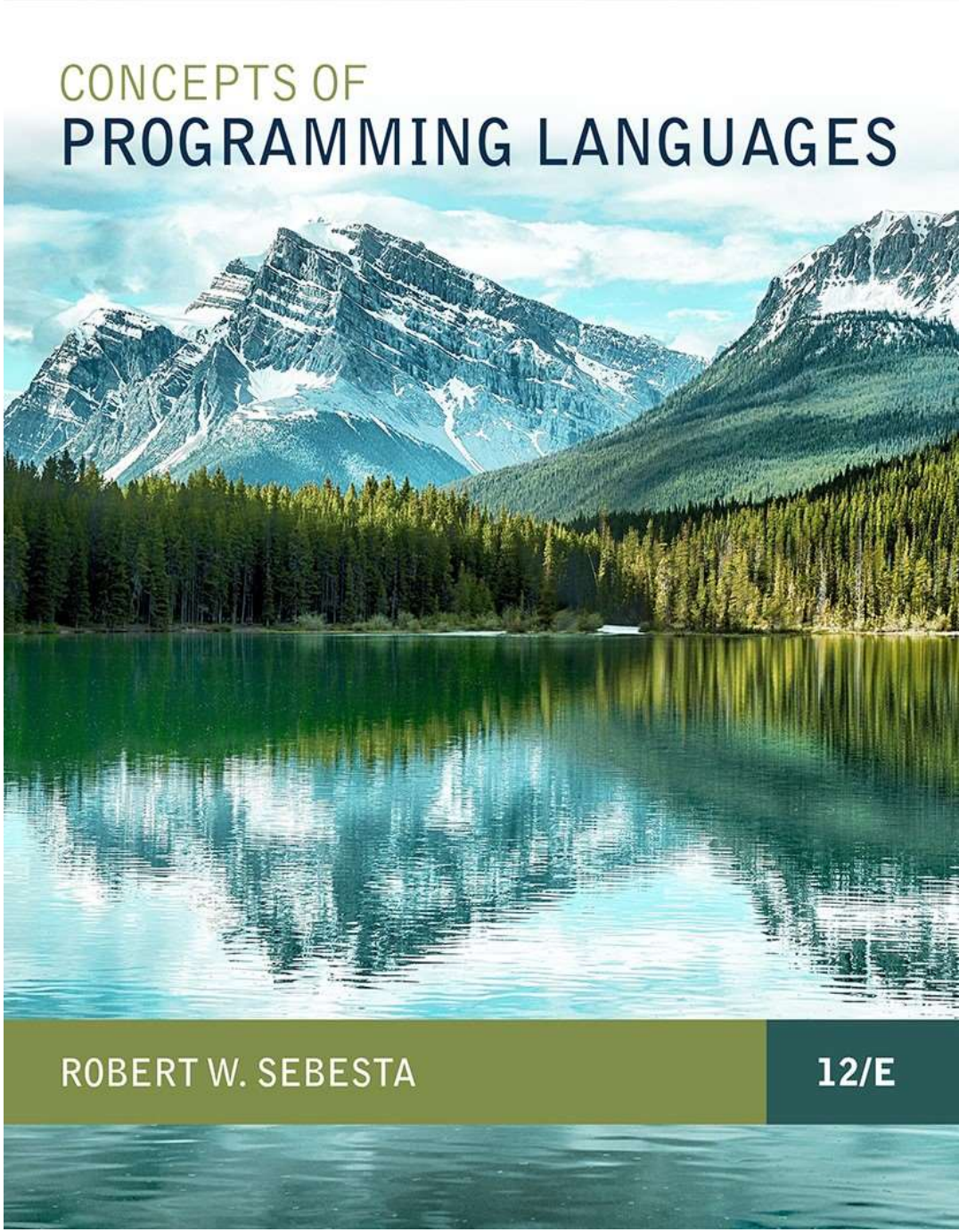
Lecture 6

Ch. 8:
Statement-Level Control Structures



ISBN 0-321-49362-1

Topics

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

Levels of Control Flow

- Control flow of programs
  - control flow (or flow of control) is the *order* in which individual statements, instructions or function calls of an imperative program are executed or evaluated.
- Levels of control flow
  - Within expressions (Lecture 7)
    - Evaluation order of expressions
  - Among program units (Lecture 9)
    - Flow of control for subprogram calls
  - Among program statements (this Lecture)
- A *control structure* is a control statement and the statements whose execution it controls

Selection Statements

- A *selection statement* provides the means of choosing between two or more paths of execution
- Two general categories:
  - Two-way selectors
    - e.g. if statements
  - Multiple-way selectors
    - e.g. switch statements

Two-Way Selection Statements

- General form:

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

- Design Issues:
  - What is the form and type of the control/condition expression?
  - How are the then and else clauses specified?
  - How should the meaning of nested selectors be specified?

The Control Expression

- The then keyword or some other syntactic marker may not used to introduce the then clause, the control expression is placed in parentheses

```
if (condition)
        { ...//stmts as then clause }
else
        {... //stmts as else clause}
```

- Condition: Boolean or arithmetic expression?
  - In C/C++ and Python the control expression can be arithmetic
  - In most other languages, the control expression must be Boolean
    - Enhances reliability

Clause Form

- Single statement or a sequence of statements?
- C-based languages

```
if (a < b)
        count += 1;
        sum += b;
else
        sum += a;
```

```
if (a < b)  {
        count += 1;
        sum += b;
} //braces required
else
        sum += a;
```

  - Braces are needed to enclose a sequence of statements in a bock
  - We refer to such statement block as a compound statement
- In Python and Ruby, clauses are statement sequences
  - e.g. Python uses indentation to define clauses

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

Then clause

Pascal:

```
if color = red then
        writeln('A red car!');
        writeln ('love it');
else
        writeln( 'Oh no! ');
```

Java:

```
if (color == red) {
        …"A red car!" …
        …"love it" …
}
else {
        …( 'Oh no! '); ..
}
```

Nesting Selectors

- Java example

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

Which if gets the else? What does Java semantics rule say?

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

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

Nesting Selectors (continued)

- Ruby
  - Statement sequences as clauses

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

- Python

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

Discussion

- Although both Ruby and Python resolved "dangling else" problem
  - identify one "downside" of Ruby's approach; explain or use an example to illustrate the problem
  - Identify one "downside" of Python's approach; explain or use an example to illustrate the problem

Selector Expressions

- In some functional languages (ML, F#, Lisp) selection can be represented as an expression (instead of a statement)
  - F# Example

```
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 a unit type, which has no value). 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: switch

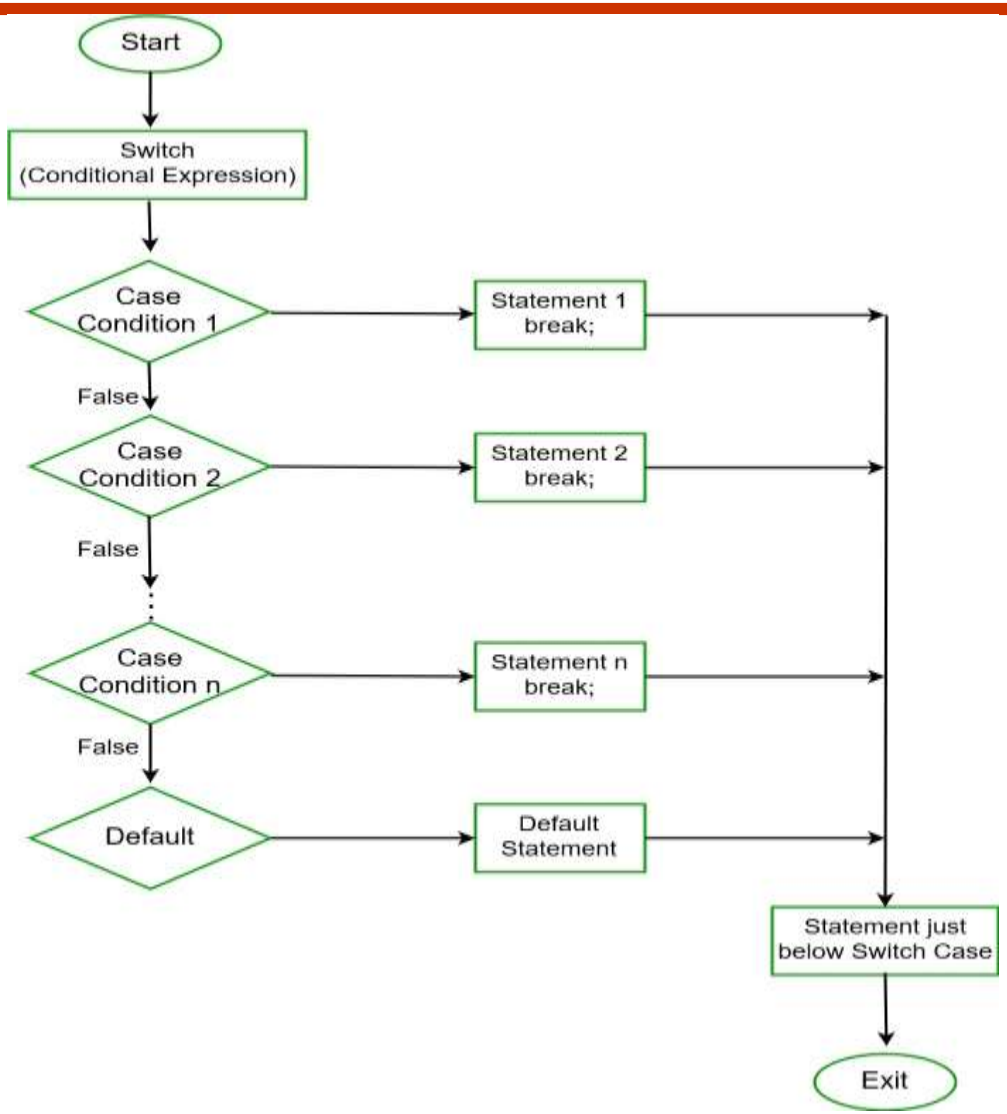
- C, C++, Java, and JavaScript:

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

Restrictions in C's switch statements:

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 or break at the end of selectable segments*)
4. default clause is for unrepresented values (if there is no default, the whole statement does nothing)

switch statement: general semantics



Practice 1

```
switch (day) {
      case 1:
            … "Monday"…;    //assume this is a println statement
      case 2:
             …"Tuesday" …;
      case 5:
            … "Friday" …;
      case 6:
      case 7:
         … ”weekend” … ;

      }
```

Questions:

(1) Would it print out “Friday” only when day is 5?
(2) What would happen if day is 4?
(3) Would it print out “weekend” for when day is 6 or 7?
(4) Find all the problems in the above code and make corrections (type up and prepare to share with the class when return to the main room);

Practice 2

```
--Ada language
case A is      -- Execute something depending on A's value:
   when 1                    => Fly;      -- if A=1, execute Fly.
   when 3 .. 10              => Put(A);   -- if A is 3 to 10, print it
   when 11 | 14    => null;          -- if A is 11 or 14, do nothing.
   when 2 | 20..30 => Swim;      -- if A is 2 or 20 through 30, Swim
   when others     => Complain; --  default. value, execute Complain.
 end case;
```

Practice: Rewrite the above case statement in Java. Enjoy!

switch statement: Enhancement

- 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)
  - Also, in C# the control expression and the case constants can be strings
- Question:
  - How does Java's switch statement differ from C's switch? From C#'s switch?

Multiple-Way Selection: case

- Similar to switch, but the keyword is "case"
- Ada has case statements
- Ruby has two forms of case statements
  - Here's one example

```
leap = case
 when year % 400 == 0 then true    #expression as selector
 when year % 100 == 0 then false
 else year % 4 == 0                #default
end
```

- Some languages such as Python doesn't have multiple-way selection statements
  - Using multiple if-clause could achieve the same goal

Implementing Multiple Selectors

- Goal: More efficient than multiple if-clause

- Approaches:
  - 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

Selective Statements: Summary

- Variety of statement-level structures
- if statements: commonly used selection structure
- switch/case statements: more efficient way in making choices with multiple selections
  - With restrictions
- Functional programming languages may use quite different control structures
  - E.g. selectors/selections can be in the form of expressions

Iterative Statements

- The repeated execution of a statement or compound statement is accomplished either by iteration or recursion
  - Iteration more efficient than recursion
- General design issues for iteration control statements:
  1. How is iteration controlled?
  2. Where is the control mechanism in the loop?
- Iterative statements: Loops
  - Counter controlled loops
  - Logically controlled loops

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?
  4. What is the value of the loop variable after loop termination?

Examples

```
for (int i=0; i<n; i++) { … }
//initial: 0; terminal: n-1; step: 1

for (int i=10; i<=100; i=i+3) { … }
//initial: 10, terminal: 100; step: 3
//Some languages include terminal value in the iteration while some not

for (int i=0; i> -50; i=i-5) { … }
//initial: 0, terminal: -50; step: -5
```

Counter-Controlled Loops: C

- C-based languages

  for ([expr_1] ; [expr_2] ; [expr_3]) statement

  – The expressions can be sequences, with the expressions/assignments separated by commas
  - The value of a multiple-statement expression is the value of the last statement in the expression
  - Any expression in loop control could be absent

    for (; ;) { } //infinite loop
- Design choices:
  - There is no explicit loop variable – thus, no fundamental difference with logically controlled loops.
  - 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

Counter-Controlled Loops: C++/Java/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)

```
for (int i=0; i< n; i++) {          //C++
        …
}       //end of i's scope
```

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

Pitfall

- A for loop in C-like languages may not serve as a counter controlled loop, e.g.

```
for (; ; ) { }  //infinite loop

for (; index < size && data[index++] != seed;) { .. }
//linear search code where expr1 and expr3 are absent
```

Counter-Controlled Loops: Examples

- Python

  `for` loop_variable **`in`** object:
  - loop body

  [**`else`**:
  - else clause]

```
for ct in range (5) :
    print(ct * 2)
else:
    print("done")
```

- 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, as in **`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
- At loop termination, the loop variable has the last value that was assigned to it
- The else clause, which is optional, is executed if the loop terminates normally

Logically-Controlled Loops

- Repetition control based on a Boolean expression
- Pretest or posttest?
  - e.g. while vs. do-while statements
  - C-based have both pretest and posttest forms

```
while (control_expr)
        loop body
```

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

  - Python has while loop, but no do-while loop.
- Java vs. C/C++
  - In C/C++ it is legal to branch (e.g. by goto stmt) into the body of a logically-controlled loop
  - In Java control expression must be Boolean and the body can only be entered at the beginning -- Java has no goto

Logically vs. Counter Controlled Loops

- Logically controlled loops are general form
  - Any counter controlled loops can be expressed by logically controlled, e.g. for (e1; e2; e3) stmt; equivalent to

```
e1;
while (e2) {
    stmt;
    e3;
}
```

  - In C-based while or do-while loops can also be expressed as for loops, e.g.

    while (e) stmt; equivalent to for (; e;) stmt;

  - In other languages such as Python counter controlled loops are restricted to step-wise iterations.

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)
- break and continue statements
  - C , C++, C#, Python etc. have unlabeled unconditional exits (break) and unlabeled control statement (continue)
    - continue statement skips the remainder of the current iteration, but does not exit the loop
  - Java has labeled versions of break and continue

```
for (i=0; i<10; i++) {
   if (i== 5) break;
}
```

```
for (i=0; i<10; i++) {
   if (i== 5) continue;
}
```

Break and Continue

```
j = 0;
while (j<10) {
        if (is_enough(j)) break;
        j += 1;
}
```

```
j = 0;
while (j<10) {
        if (is_enough(j)) continue;
        j += 1;
}
 //how many iterations maximum?
```

```
int is_enough (j) {
    //assume VAL is a know constant
    if (j == VAL ) return true;
    else return false;
}
```

Labelled Break Statement – Java

```
//Example 1 – simple case but improves readability
int i=7;
loop1:
while(i<20) {
        if(i==10)
                break loop1;
        System.out.println("i ="+i);
        i++;
}
System.out.println("Out of the loop");
```

//Example 2: used in nested loops

```
int[][] arr = { { 1, 2 }, { 3, 4 }, { 9, 10 }, { 11, 12 } };
boolean found = false;
int row = 0, col = 0;
searchint:              // find index of first int greater than 10
for (row = 0; row < arr.length; row++) {
        for (col = 0; col < arr[row].length; col++) {
                if (arr[row][col] > 10) {
                        found = true;
                        break searchint;
                } //end if
        } //end for col
} //end for row
if (found)
        print("First int greater than 10 is found at index: [" + row + "," + col + "]")
```

Detecting abnormal loop break out

```
#Python
j = 0
s = an random integer
n = array size
while j< n :
      if arr[j] == seed):
            print (s, “found in pos”, j)
            break  #match
      j += 1
else : #loop terminate normally
      print(s, “not in loop”)
```

```
#Python

for j in range (n) :
    if arr[j] == s :
          … found …
else :
 #note else matches for, not if
    … not in list …
```

Iteration Based on Data Structures

- The number of elements in a data structure controls 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
  - Example: Java 5.0 (uses for, although it is called foreach

```
for (String name : nameLst) {
        System.out.println(name);
}
```

  - Java 8.0 has forEach

```
nameLst.forEach(name -> {
        System.out.println(name);
});
```

Python's yield statement

Suppose we have a function, traverse, that produces the nodes of a data structure

```
def traverse(self):
  yield nod
```

- Every time traverse is called, it returns the next node of the structure on which it is called – yield is like a return
- However, yield remembers its state – it starts in the state it was in at the end of its last execution. It is history sensitive.
- In Python, we call such traverse function as "generator".
  - e.g. generating an infinite_sequence

```
def infinite_sequence():
    num = 0
    while True:
        yield num
        num += 1
```

```
>>> gen = infinite_sequence()
>>> next(gen)
0
>>> next(gen)
1
>>> next(gen)
2
>>> next(gen)
3
```

Unconditional Branching

- Transfers execution control to a specified place in the program
  - Most popular: the goto statement
- Represented one of the most heated debates in 1960's and 1970's
- Major concern: Readability
- Contemporary programming languages/paradigms support limited use only of goto or break
  - Some languages do not support goto statement (e.g., Java)
  - Some restrict the usage within switch statement or certain user-defined loops only (C#)
- Loop exit statements are restricted and somewhat camouflaged goto's

Iterative Statements: Summary

- Variety of statement-level structures
  - Most popular: selections and iterations
- 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
  - To be discussed in Lectures 15 & 16