

## Chapter 8

# Statement-Level Control Structures

### 8.1 Introduction

### 8.2 **Selection** Statements

### 8.3 **Iterative** Statements

### 8.4 Unconditional **Branching**

### 8.5 **Guarded Commands**

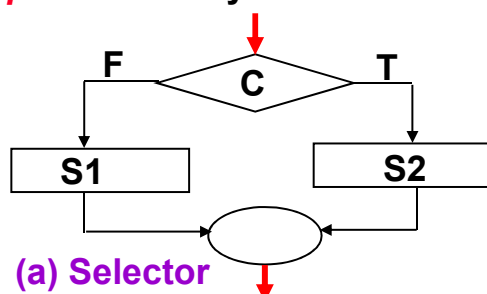
*The flow of control, or execution sequence, in a program can be examined at several levels.*

- flow of control **within expressions** : -> operator **precedence** and **associativity**
- flow of control **among statements** : -> **statement-level control** structures
- flow of control **among unit** : -> **procedure invocation**

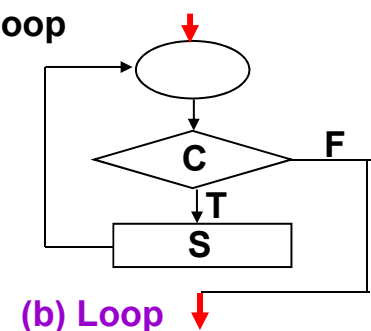
*We have the theoretical result that only **sequence**, **selection**, and **pretest logical loops** are absolutely required to express computations*

## 8.1 Introduction

- **Computations in imperative languages**
  - evaluating expressions and assigning resulting values to variables
    - ⇒ choosing among alternative control flow paths
    - ⇒ causing the repeated execution of certain collection of statements
- The **control statements** of early FORTRAN
  - directly related to machine language instructions
- It was proven that **all algorithms that can be expressed by flowcharts** can be coded in a programming language with only two control statements ;
  - one for **choosing** between two control paths
  - one for logically controlled **iterations**
- **Writability** is enhanced by a large number of control statements
  - How much should a language be expanded to increase its **writability**, at the expense of its **simplicity and size** ?
- **Flow graph** of two-way selector and a logically controlled loop



Single-Entry  
Single-Exit



## • Compound Statements

- In **Algol 60**,

⇔ **compound statement** : a collection of statements to be abstracted to a single statement

```
begin
    statement-1 ;
    statement-2 ;
    .....
end
```

⇔ **block** : compound statement + data declarations

```
begin
    integer index, count ;
    statement-1 ;
    statement-2 ;
    .....
end
```

- **Pascal** follows ALGOL 60's design for compound statements, but **does not allow block**

- The **C** language uses the **braces** to delimit both compound statements and blocks

- **Design Issues**

A **control structure** is a control statement and the statements whose execution it controls

⇔ **Can the control structure have multiple entries ?**

⇒ It is now generally believed that multiple entries add little to the flexibility of control structures, relative to the decrease in readability caused by increased complexity

## 7.3 Selection Statement

- It provides the means of **choosing between two or more execution paths** in a program

⇔ **two-way selection, n-way selection** (multiple selection)

### (1) Two-Way Selection Statements

- **Design Issues**

- **What is the form and type of the expression that controls the selection ?**

⇔ In most of the languages : **boolean expression**

⇔ In **C** : **arithmetic expression**

- **Can a single statement, a sequence of statements, or compound statement be selected ?**

- **How should the meaning of selectors nested in then clauses of other selectors be specified ?**

⇔ **by syntax, or by semantic rule**

## • Single-Way Selectors

- All imperative languages include a single-way selector, in most cases as *a subform of a two way selector*. Two exceptions are BASIC and FORTRAN
- In **FORTRAN** (a logical IF),
  - ⇔ IF (*Boolean expression*) *statement*
  - ⇒ The selector control expression is *boolean type*, and only *a single statement is selectable*
  - ⇒ Nesting is not allowed
  - ⇒ promotes the use of goto statement
  - ⇒ *very simple and highly inflexible*
- The *compound statement* provides the selection construct with a simple mechanism for conditionally executing *groups of statements*. In **ALGOL 60**,

```
if (Boolean expression) then
    begin
        statement-1 ;
        statement-2
        ....
        statement-n
    end
```

- Most of the languages that followed ALGOL 60, including FORTRAN 77 and 90, provide single-way selectors that can select a compound statement or a sequence of statements

## • Two-Way Selectors

- It allows one of *two control paths to be selected*
- In **ALGOL 60**,

```
if (Boolean expression)
    then (compound) statement
    else (compound) statement
```

## • Nesting Selectors

⇔ *Ambiguosness* in nested selectors

```
if (sum = 0) then
    if (count = 0)
        then result := 0
    else result := 1
```

- *In most imperative languages, the static semantics of the language* specify that the else clause is always paired with *the most recent unpaired then* clause
- In **ALGOL 60**,
  - ⇔ an if statement is not allowed to be nested directly in a then clause
  - ⇒ must be placed in compound statement

```
if (sum = 0) then
    begin
        if (count = 0)
            then result := 0
            else result := 1
        end
    end
```

```
if (sum = 0) then
    begin
        if (count = 0)
            then result := 0
        end
    else result := 1
```

- If the last clause in an *if*, whether *then* or *else*, is not a compound, there is no syntactic entity to mark the end of the whole selection construct

⇒ use of *special word*

⇒ in Modula-2, *END*

⇒ in FORTRAN 77, *END IF*

⇒ In Modular-2

```
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
```

⇒ in **Python**

⇒ uses *indentation* to define clauses

```
>>> if 1 + 1 == 2:
...     print "foo"
...     print "bar"
...     x = 42
```

Because *blocks are denoted by indentation* in Python, indentation is uniform in Python programs. And indentation is meaningful to us as readers.

```
>>> if 1 + 1 == 2:
...     print "foo"; print "bar"; x = 42
```

```
>>> if 1 + 1 == 2: print "foo"; print "bar"; x = 42
```

```
>>> if foo:
...     if bar:
...         x = 42
... else:
...     print foo
```

## (2) Multiple Selection Constructs

- It allows the selection of *one of any number of statements or statement groups*
- a generalization of a selector

### • Design Issues

- What is the *form and type of the expression* that controls the selection ?
- Can a *single statement, a sequence of statements, or a compound statement* be selected ?
- Should the entire construct be *encapsulated in a syntactic structure* ?
- Should execution flow through the structure be restricted to include just a single selectable statement ?
- How should *unrepresented selector expression values* be handled, if at all ?

### • Early Multiple Selector

- In FORTRAN,

```
GOTO integer_variable, (label-1, label-2, ..., label-n)
```

```
GOTO (label-1, label-2, ..., label-n), expression
```

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

3-way selector  
(arithmetic IF)

*Multiple Entry  
(lack of encapsulation)*

```
10  IF (expression) 10,20,30
...
...
20  GO TO 40
...
30  GO TO 40
...
40  ....
```

*if it is omitted,  
error?*

- **Modern Multiple Selector : Case**

- In **ALGOL-W**,

- ⇒ The structure is **encapsulated**, and provide a **single selectable segment**
- ⇒ The executed statement is the one chosen by the value of the expression

```
case integer_expression of
begin
    statement-1 ;
    .....
    statement-n
end
```

- In **Pascal**,

- ⇒ selectable segments are **labeled**
- ⇒ the expression is of **ordinal type** (integer, Boolean, character, or enumeration type)
- ⇒ Semantics : the expression is evaluated, and the value is compared with the constants in the constant lists
- ⇒ the constant lists must be of **the same types as the expression**, and they must be **mutually exclusive, but need not to be exhaustive**

**implicit branching**

```
case expression of
    constant_list_1 : statement_1 ;
    ....
    constant_list_n : statement_n ;
end
```

mutually  
exclusive

```
case index of
    1, 3 : begin
        odd := odd + 1 ;
    end
    2, 4 : begin
        even = even + 1 ;
    end
    else writeln ("Error in case") ;
end
```

ANSI/IEEE  
Pascal Standard

- In **C (switch)**

- ⇒ the control expression and constant expressions are **integer type**
- ⇒ It **does not provide implicit branches** at the end of those code segments (**reliable** vs. **flexible**)

```
switch (expression) {
    case constant_expression-1: statement-1 ;
    ....
    case constant_expression-N: statement-N ;
    default : statement-N+1 ;
}
```

```
switch (index) {
    case 1 :
    case 3 : odd += 1 ;
        break ;
    case 2 :
    case 4 : even += 1 ;
        break ;
    default : printf("Error in switch");
}
```

**explicit  
branching**

- In **Python**,

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

## 8.3 Iterative Statements

- **Iterative Statement**

- It is one that causes a statement or collection of statements **to be executed zero, one, or more times**
- It is often accomplished in a functional language by **recursion** rather than by iterative constructs
- The first iterative constructs in programming languages were directly related to arrays
- Design Issues
  - ⇒ **How is the iteration controlled ?**
    - ⇒ logical, counting, or a combination of two
  - ⇒ **Where should the control mechanism appear in the loop ?**
    - ⇒ **pretest, posttest, or user defined**

### (1) Counter-Controlled Loops

- **Loop variable** in which the count value is maintained
- **Loop parameters**
  - ⇒ **initial, terminal, stepsize**

```
DO 30 I=1,100,2
...
30 CONTINUE
```

- Counter-controlled loops are often supported by machine instruction

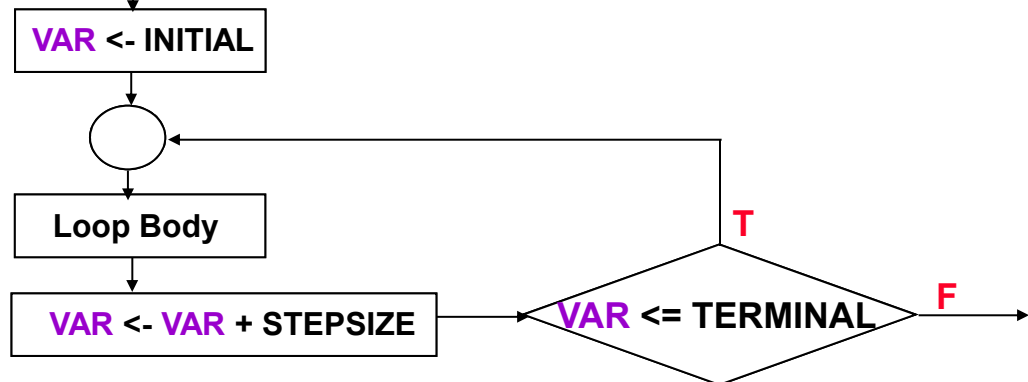
- **Design Issues**

- **What is the type and scope of the loop variable ?**
  - ⇒ integer, character, enumeration, floating types
- **What value does the loop variable have at loop termination ?**
- **Should it be legal for the loop variable or loop parameters to be changed in the loop, and if so, does the change affect loop control ?**
- **Should the test for completion be at the top or the bottom of the loop ?**
- **Should the loop parameters be evaluated only once, or once for every iteration ?**

## • The FORTRAN IV DO

```
DO 30 I=1,100,2
...
30 CONTINUE
```

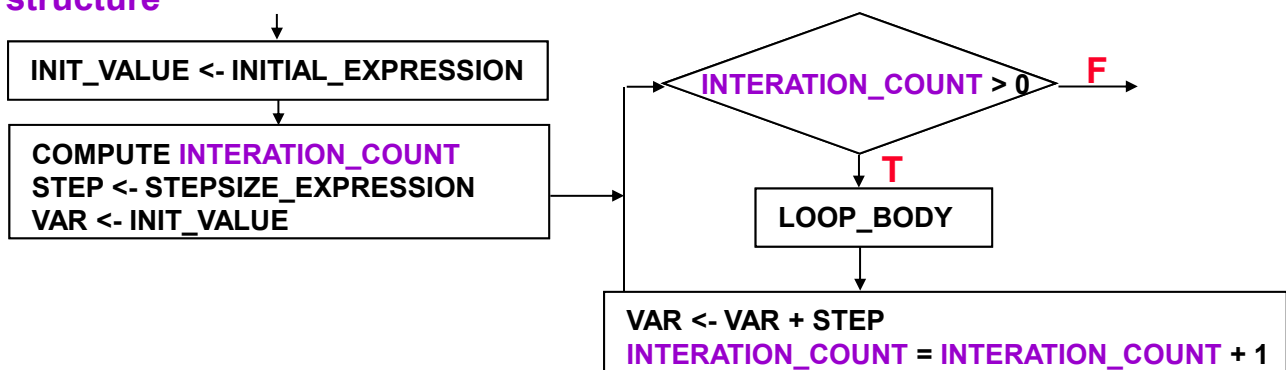
- **posttest**
- the initial, termination, and stepsize parameters are restricted to **unsigned integer constants**, or **simple integer variable** with positive values
- the value of loop variable is
  - ⇒ **undefined** upon normal loop termination
  - ⇒ **its most recently assigned value** up abnormal termination
- the loop variable and loop parameters **can not be changed in the loop body**, so there is no reason to evaluate the loop parameters more than once
- extended loop body : by GO TO



## • The Do statement of FORTRAN 77 and FORTRAN 90

```
Do label variable = initial, terminal [,stepsize]
```

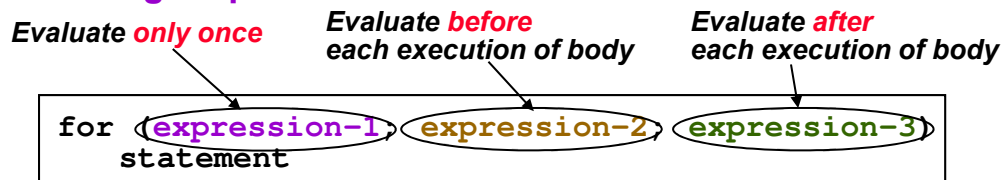
- **pretest**
- the loop variable can be **integer**, **real**, or **double-precision type**
- the loop parameters are allowed to be **expressions** and can have positive and negative values
- the loop is controlled by **iteration count**, not the loop parameters, so even if the parameters are changed in the loop, which is legal, those change can not affect loop control
  - ⇒ the iteration count is an **internal variable** that is inaccessible to the user code
- DO construct can be entered only through the DO statement - **sing-entry structure**





- The **C** **for** statement

- a **pretest counting loop** structure



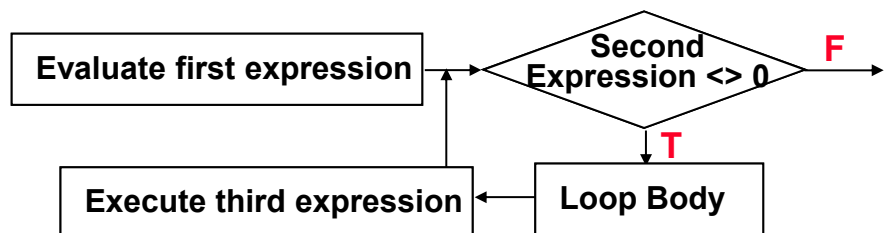
- If the value of the second expression is **zero**, the **for** is **terminated**; otherwise, the loop statements are executed

```
for (index = 0 ; index <= 10; index++)
    sum = sum + list[index] ;
```

- All of the expressions of **C**'s **for** are optional

```
for (sum = 0.0, count = 0 ;
    count <= 10 && sum < 1000.0;
    sum = sum + count++) ;
```

- there is no explicit loop variable
- all involved variables can be changed in the loop body



- In **Python**,

```
#!/usr/bin/python
for letter in 'Python':    # First Example
    print 'Current Letter :', letter
fruits = ['banana', 'apple', 'mango']
for fruit in fruits:      # Second Example
    print 'Current fruit :', fruit
print "Good bye!"
```

```
Current Letter : P
Current Letter : y
Current Letter : t
Current Letter : h
Current Letter : o
Current Letter : n
Current fruit : banana
Current fruit : apple
Current fruit : mango
Good bye!
```

```
#!/usr/bin/python
fruits = ['banana', 'apple', 'mango']
for index in range(len(fruits)):
    print 'Current fruit :', fruits[index]
print "Good bye!"
```

```
Current fruit : banana
Current fruit : apple
Current fruit : mango
Good bye!
```

```
#!/usr/bin/python
for num in range(10,20):    #to iterate between 10 to 20
    for i in range(2,num):
        if num%i == 0:      #to determine the first factor
            j=num/i          #to calculate the second factor
            print '%d equals %d * %d' % (num,i,j)
            break            #to move to the next number, the #first FOR
    else:                    # else part of the loop
        print num, 'is a prime number'
```

```
10 equals 2 * 5
11 is a prime number
12 equals 2 * 6
13 is a prime number
14 equals 2 * 7
15 equals 3 * 5
16 equals 2 * 8
17 is a prime number
18 equals 2 * 9
19 is a prime number
```

- the **else** statement is executed when the loop has exhausted iterating the list.
- The **else** clause, which is optional, is executed if the loop terminates normally



## (2) Logically controlled loops

- the repetition control is based on a Boolean expression

### • Design Issues

- Should the control be *pretest or posttest* ?
- Should the logically controlled loop be a special form of counting loop or a separate statement ?

### • Examples

- Some imperative languages - for example, Pascal, Modula-2, and C - include both pretest and posttest logically controlled loops
- In C,

*pretest*

```
while (expression) statement

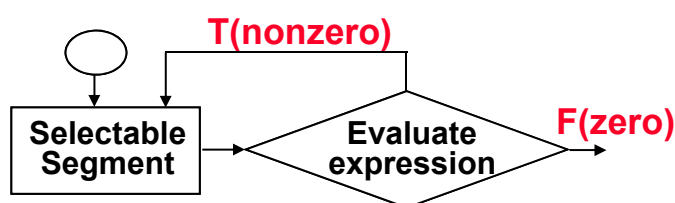
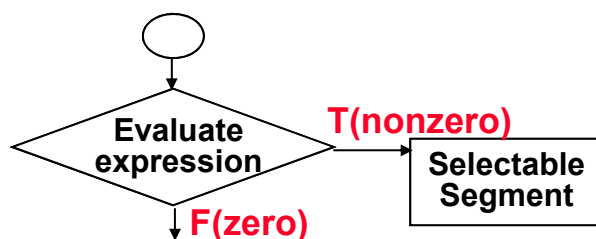
scanf ("%d", &indat) ;
while (indat >= 0) {
    sum = sum + indat ;
    scanf ("%d", &indat) ;
}
```

*posttest*

*body is execute at least once*

```
do statement while (expression)

do {
    indat = indat / 10;
    digits = digits + 1;
} while (indat > 0);
```



## (3) User-Located Loop Control Mechanisms

- a programmer chooses 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 the mechanism be allowed to appear in a controlled loop or only in one without any other control ?
- Should only one loop body exit, or can enclosing loops also be exited ?

### • In Ada,

*loop can be labeled*

```
OUTER_LOOP :
  for ROW in 1..MAX_ROWS loop
    INNER_LOOP :
      for COL in 1..MAX_COLS loop
        SUM := SUM + MAT(ROW, COL);
        exit OUTER_LOOP when SUM > 100.0;
      end loop INNER_LOOP
    end loop OUTER_LOOP
```

*infinite loop*

```
loop
  ...
  SUM := SUM + INDEX ;
  exit when sum >= 1000;
  ...
end loop
```

### • In C,

*skip the remaining loop body*

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

**Multiple Exit**

*terminate the loop*

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

## (4) Iteration based on Data Structure

- Iteration based on data structure

- The loops are controlled by ***the number of elements in a data structure***, rather than counter or boolean expression

- Example

- Java 5.0

⇒ the enhanced version of `for` simplifies iteration through ***the value in an array*** or ***the objects in a collection*** that implements the `Iterable` interface

⇒ example

⇒ if we had an `ArrayList` collection named `myList` of strings, the following statement would iterate through all of its elements, setting each to `myElement`

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

- C#

```
String[] StrList={"Bob", "Carol", "Ted", "lala"};  
...  
foreach (String name in StrList) ... ;
```

## 8.4 Unconditional Branching

- An ***unconditional branch statement*** transfers execution control to a specified place in the program

### (1) Problems with Unconditional Branching

- The unconditional branch, or ***goto***, is ***the most powerful statement*** for controlling the flow of execution of a program statements, but it is this very power that makes its use ***dangerous***
  - readability is best when the execution order of the statements is nearly the same as the order in which they appear - this usually means ***top to bottom***
  - a few languages have been designed without a `goto`, for example, ***Java***
  - However, ***most currently popular languages include a goto statement***
  - the languages that have eliminated the `goto` have provided additional control statements, usually in the form of loop and subprogram exits, to replace many of the typical application of the `goto`

## (2) Label Forms

### • Label Forms

- In **ALGOL 60** and **C** : use their identifier forms for labels
- In **FORTRAN** and **Pascal** : use unsigned integer constants for labels
- In **PL/1** : allows the labels to be variables

### • Restrictions on Branches

→ *most language restricts their usages*

- In Pascal,

- ⇒ Pascal labels must be declared as if they were variables, but they cannot be passed as parameters, stored, or modified
- ⇒ a *goto* can never have as its target a statement in a compound statement of a control structure, unless execution of that compound statement has *already begun and has not yet terminated*

illegal

```
while ... do begin
100:   ...
      ...
end
while ... do begin
      ...
      goto 100 ;
      goto 200 ;
end
while ... do begin
200:   ...
end
```

legal

```
while ... do begin
100:   ...
      ...
      while ... do begin
            ...
            goto 100 ;
            ...
      end ;
end ;
```

⇒ Pascal allows to branch to a different procedure (error-propagation)

## 8.5 Guarded Commands (by Dijkstra 1975)

### • Dijkstra's selection construct

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

- all the Boolean expressions are evaluated each time the construct is reached during execution. If more than one expression is true, one of the corresponding statement is *nondeterministically chosen* for execution. If none is true, a *run-time error* occurs that causes program termination

⇒ It forces the programmer to consider and list all possibilities (*exhaustive listing*)

- In **Ada** case statement,

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

*if (i=0) and (j=1) ?  
if (i=j) and (i<>0) ?*

### • Dijkstra's loop structure

```
q1 := Q1; q2 := Q2; q3 := Q3; q4 := Q4;
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
```

- the concurrency control in the Ada languages

