

Ch. 7: Control Structures

I. Introduction

- A. Flow of control can be at multiple levels: within expressions, among statements (discussed here), and among units.
- B. Computation in imperative languages uses assignment statements with the addition of: (Bohm & Jacopini, 1966):
 - 1. Selection statement
 - 2. Repeated execution (sequence is a special case of this)
- C. Initial control statements reflected underlying architecture
 - 1. FORTRAN's arithmetic if
 - 2. VAX post-loop condition check for loop counter
- D. Agreement that control structures should have single entries and exits

II. Compound Statements

- A. Missing in early FORTRAN, added in ALGOL 60:

```
begin
statement_1;
...
statement_n;
end
```

These allow a collection of statements to be abstracted into a single statement

- B. Adding data declarations to a compound statement makes it a block
- C. Issue for both selection and iteration control statements: whether there are multiple entry points

III. Selection Statements: Allows for choosing between execution paths in a program

- A. Two-Way Selection Statements

- 1. Design Issues

- a. What is the form and type of the expression that controls the selection?
- b. Can a single statement, sequence of statements, or a compound statement be selected? (single statement selection leads to gotos.)
- c. How does a "else" match up with previous "if" statements?

- 2. Single-way Selectors: FORTRAN (logical if) & BASIC

- a. IF (Boolean expr.) statement

In this FORTRAN selector the statement may not be compound and logical ifs may not be nested. Would need goto, as in:

```
IF (.NOT. condition) GO TO 20
I = 1
J = 2
20 CONTINUE
```

Since the code can contain multiple labels, it can contain multiple entry points.

b. Compound statement in ALGOL 60 adds the capability:

```
if (Boolean expr) then
  begin
    statement_1;
    ...
    statement_2;
  end
```

3. Examples of Two-Way Selectors

a. ALGOL 60 was the first, with the form:

```
if (Boolean expr)
  then statement
  else statement
```

Where both statements could be compounded

4. Nesting Selectors: which "if" does an "else" go with?

a. Example:

```
if (cond1.) then
  if (cond2.) then
    statement1;
  else
    statement2;
```

Which if does the "else" go with?

b. Static semantics: Language definition tells which it should be. E.g. in Pascal: "else" goes with most recent unpaired "if"

c. Syntax can be used instead: E.g. ALGOL 60 where an if may not be directly nested in a "then" clause, but must be in a compound statement::

```
if (cond1.) then
  begin
    if (cond2.)
      then statement1
      else statement2
  end
```

or else as:

```
if (cond1.) then
  begin
    if (cond2.) then statement1
  end
else statement2
```

Same thing in Pascal & C

5. Selection Closure: an alternative to the above compound statement for nested "else" is to use special words to mark the end of an if.

a. E.g. in Ada:

```
if cond1 then
    statement1;
else
    statement2;
end if;
```

and also:

```
if cond1 then
    statement1a;
    statement1b;
else
    statement2a;
    statement2b;
end if;
```

Note that inside the "then" and "else" clauses we have statement sequences rather than compound statements. The "end if" marker allows this.

b. The previous nested "else" examples would then look like:

```
if cond1. then
    if cond2. then
        statement1;
    else
        statement2;
    end if;
end if;
```

Or as:

```
if cond1. then
    if cond2. then
        statement1;
    end if;
else
    statement2;
end if;
```

B. Multiple Selection Constructs

- generalization of a selector, & can be built with multiple selectors.

- An additional construct is more clear, however.

1. Design Issues

a. What is the form and type of the expression controlling selection?

b. May a single statement, a sequence of statements, or a compound statement be selected?

c. Is the entire construct encapsulated in a syntactic structure?

d. Is execution flow limited to a single selectable statement?

e. Is there a "default" clause for unrepresented selector values?

2. Early Multiple Selectors in FORTRAN: three-way selector (arithmetic if)

a. IF (arithmetic expression) N1, N2, N3

where the three labels are statements corresponding to the expression values of negative, 0, positive respectively. It would often be used as:

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

Problem: allows multiple entry & exit points using labels and gotos

b. First true multiple selection statement: again in FORTRAN:

```
          GO TO (label 1, label 2, ... label n), expression
```

where the integer result of evaluating "expression" functioned as an index into the branch labels (1 went to the first, etc.)

3. Modern Multiple Selectors

a. ALGOL-W (1966):

```
case integer-expression of
begin
statement1;
...
statementn;
end
```

where statements could be compound, and only one is chosen.

b. Pascal:

```
case expression of
constant_list1: statement1;
...
constant_listn: statementn;
end
```

Which functions similarly to ALGOL-W, except expression is of ordinal type (integer, boolean, character, enumerated type), and selectors are lists.

- (1). Only one of the selectors is executed, constant lists must be mutually exclusive, and the constant lists' union need not be exhaustive. It has single entry & exit point.
- (2). Issue: what about unrepresented selector options? An "else" clause was added to take care of this.

c. C mutiple selector

```
switch (index) {
    case 1:
    case 3: statement1;
           statement2;
           break;
    case 2:
    case 4: statement3;
           statement4;
           break;
    default: statement5;    //possibly an error message?
}
```

Note that without a break control flow continues to the next statement.

- d. Ada requires that the union of the constant lists be exhaustive, and also provides an "others" clause to help ensure this.
- e. In cases where the selectors are not integer or an ordinal type, nested if statements must be used, as in Ada's:

```
if score >= 90 then grade := 'A';
elseif score >= 80 then grade := 'B';
elseif score >= 70 then grade := 'C';
end if;
```

IV. Iterative Statements:

- cause a statement or collection of statements to be executed 0, 1, or more times. Thus, sequence is a special case of iteration.

A. Counter-Controlled Loops

1. loop parameters: loop variable, initial value, terminal value, stepsize
2. Design Issues
 - a. What is the type and scope of the loop counting variable?
 - b. Is the loop variable defined upon exiting the loop?
 - c. Can the loop variable or loop parameters be changed inside the loop?
 - d. Is the test for completion at the top or the bottom of the loop?
 - e. Are the loop parameters evaluated on every iteration?

3. FORTRAN IV DO loop

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

where loop parameters must be unsigned integer constants or positive integers (no descending loops). Loop variable is undefined upon exiting loop, but is defined upon abnormal termination. Loop parameters may not be changed inside the loop, so loop parameters evaluated only once. Allows multiple entry and exit points to the loop. It is a posttest loop construct.

4. FORTRAN 77 & 90 DO loop: same as FORTRAN IV except that:

- a. loop variable is allowed to be integer, real, or double; loop parameters can be expressions and can have negative values; loop parameters are evaluated at the beginning of the execution to give an iteration count, after which loop parameters may be changed inside the loop without affecting the loop control.
- b. comma optionally added after the label

```
DO 10, K = 1, 10
```

So that "DO 10 K = 1, 10" with an inadvertant decimal replacing the comma would not become the assignment statement:

```
DO10K = 1.10
```

- c. loop completion test moved from posttest to pretest
 - d. FORTRAN 90 added an "END DO [name]"
5. ALGOL 60 for statement: flexibility led to excessive complexity
- a. syntax:

```

<for_stmt> -> for var := <list_element> {, <list_element>} do <st
ment>
<list_element> -> <expression>
                | <expression> step <expression> until <expression>
                | <expression> while <Boolean_expr>

```

Note that this can combine counter control with a boolean expression [See overhead]

- b. loop parameters evaluated on each iteration, allowing step size to change at each iteration. E.g.

```

i := 1;
for count := 1 step count until 3 * i do
    i := i + 1

```

Note that the first time through the loop count is assigned one and so stepsize is one. The second time through the loop count has been incremented by stepsize (one) and so is now two. This means that "step count" changes the stepsize to two. Loop does eventually terminate since count increases faster than (3 * i), as it doubles while (3*i) increases by 3 each time.

- c. Design choices: loop variable can be integer or real, and has most recent value upon loop termination. Loop parameters (but not loop variable) may be changed in the loop body. It is single-entry construct. It is a pretest loop.

6. Pascal for statement

- a. Syntax:

```

for variable := initial_value (to | downto) final_value do
    statement

```

- b. Design choices: loop variable must be ordinal type; loop variable undefined upon normal termination. Loop variable may not be changed inside loop. Initial and final values may be changed inside loop, but they are evaluated only once. Completion test is at top of loop.

7. Ada for statement: similar to Pascal

- a. Syntax:

```

for variable in [reverse] discrete_range loop
    ...
end loop

```

- b. Design choices: loop variable is implicitly declared at loop beginning and implicitly undeclared at loop end. E.g.

```

COUNT : FLOAT := 1.35;
for COUNT in 1..10 loop
    SUM := SUM + COUNT;
end loop

```

Where the loop "COUNT" masks the outer "COUNT"

8. C & C++ for statement

a. Syntax:

```
for (initial_exp.; continuing_cond_expression; post_loop_exp)
    statement
```

Expressions in C can be statements and return a value. Zero value means False, a non-zero value means True.

b. Example:

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

c. All expressions in the "for" are optional. It is legal to branch into a "for" loop.

d. Multiple statements may be used in a single expression in a "for" loop, separated by commas. E.g.

```
for (count = 0, sum = 0; x >0; scanf("%d", x), sum += x, count++)
    ;
```

e. In C++, the declaration of the loop counter has scope until the end of the function. E.g.

```
for (int count = 0; count < length; count++) { ... }
```

which is different from:

```
{ int count;
  for (count = 0; count < length; count++) { ... }
}
```

f. Note that C's "for" loop can both count and contain logical tests

B. Logically Controlled Loops

1. Design Issues

a. Is the control pretest or posttest?

b. Is the loop simply a special case of a counting loop?

2. Examples

a. C:

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

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

C. User-Located Loop Control Mechanisms (explicitly exiting loop): design issue: should only one loop be exited, or can enclosing loops also be exited?

1. Ada:

```
loop
...
end loop
```

which can contain an "exit" statement which may be conditional or unconditional:
exit [loop_label][when condition]

For instance:

```
loop
...
if SUM >= 1000 then exit;
{Could also be: exit when SUM >= 1000;}
...
end loop;
```

Multiple enclosing loops may be exited using a label:

```
OUTER:
    for ROW in 1 .. MAX_ROWS loop
INNER:
        for COL in 1 .. MAX_COLS loop
            SUM := SUM + MAT(ROW, COL);
            exit OUTER when SUM > 1000;
        end loop INNER;
    end loop OUTER;
```

2. C: Uses continue to repeat the loop with the next iteration. E.g.

```
while (sum < 1000) {
    scanf("%d", &value);
    if (value < 0) continue; {goto the top of the loop again}
    sum += value;
}
```

and also the break statement to exit the loop entirely:

```
while (sum < 1000) {
    scanf("%d", &value);
    if (value < 0) break; {leave immediate enclosing loop}
    sum += value;
}
```

D. User-Defined Iteration Control

- Rather than use a counter, use the number of elements in a user-defined structure. This is known as an *iterator*

1. LISP: the *dolist* function iterates on simple lists

2. Iterator to traverse a tree, example in C:

```
for (ptr = root; ptr; traverse( ptr)) {
    ...
}
```

V. Unconditional Branching

A. Problems with Unconditional Branching

1. All other control structures can be rewritten with a selector statement and a goto, however goto's are considered dangerous (Dijkstra's 1966: "Goto considered harmful")
2. Donald Knuth argues that there are occasions where it is useful
3. A few languages have been designed without it.

B. Label Forms

1. Ada: uses its identifier form unless it is on a line with other instructions, in which case it gets double brackets:

```
goto FINISHED;
...
<<FINISHED>>
```

2. PL/I allows the target of a branch to be a variable, so that at runtime you could branch anywhere.

C. Restrictions on Branches: In Pascal gotos are allowed but are restricted. The target of a goto can never be in a compound statement of a control structure unless execution of that compound statement has already begun and has not terminated.

1. Examples of goto that are not allowed:

```
while cond1. do
  begin
    while cond2. do
      begin
        100: ...
        ...
      end;
    while cond3. do
      begin
        goto 100;
        goto 200
        while cond3 do
          begin
            200: ...
            ...
          end;
        end;
      end;
    end;
  end;
```

2. Examples of legal goto's in Pascal:

```
while cond1 do
  begin
    100: ...
    ...
    while cond2 do
      begin
        goto 100;
        ...
      end;
    end;
  end;
```

```
procedure sub1;
  label 100;
  ...
  procedure sub2;
    goto 100
    ...
  end;
  ...
  100: ...
  ...
end;
```

3. Example where a goto seems to help the code's readability: (a letter to the editor of CACM by Rubin in 1987). This code finds the first row of an $n \times n$ integer matrix that has all zeros.

```
for i := 1 to n do
  begin
    for j := 1 to n do
      if x[i, j] <> 0 then
        goto reject;
      writeln ('First all-zero row is: ', i);
      break;
    reject:
  end;
```

VI. Guarded Commands (Dijkstra, 1975).

- This format is particularly important for specifying concurrent programming.

A. if statement using *fatbars*:

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

where all the boolean expressions are evaluated each time. If more than one is true, one of the true ones is chosen nondeterministically. If none is true, it gives a run-time error.

1. Another example:

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

Note that if $i = 0$ and $j > i$, then either the first or third is chosen nondeterministically.

If $i = j$ but $i \neq 0$ then a run-time error occurs since no condition is true.

2. E.g. a do loop using guarded commands:

```
do <boolean expression> -> <statement>
[] <boolean expression> -> <statement>
[] ...
[] <boolean expression> -> <statement>
od
```

where all the boolean expressions are evaluated on each iteration. If more than one boolean expression is true, one is chosen nondeterministically, after which the expressions are again evaluated (and so on...).

3. E.g. sorting the four variables $q_1, q_2, q_3,$ & q_4 into ascending order

```
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
```

VII. Conclusions

- A. Only selection, loops, and sequence (a special kind of loop) are absolutely required.
- B. Fundamental question: should the size of a language be minimized?
- C. Note that the functional and logical programming mechanisms are quite different from the imperative language control structures presented here.