Chapter 6

(6.2 – 6.5)

Sequence, Selection, and Iteration

Sequential Control

Assignment

- Changes the state of memory (modify variable contents)
- Does not interfere with normal flow of control
- Control moves sequentially to the next statement

C Assignment Problem

Because C does not have a boolean data type and assignment is an expression, the following is possible

```
if (x = y) { … }
```

When will the condition succeed? Fail?

Control Flow Instructions

Do not change the state of memory

Directs the thread of computation

- What hardware manages the point of control?

Low level control flow uses the *goto* instruction (machine/assembler)

The *GOTO* Instruction

Forward *goto* skips code

Backward *goto* repeats code

Unconditional vs. Conditional *goto*

The *goto* was very controversial in the days of structured programming

Goto Is Bad?

In 1968, Dijkstra wrote the paper
"Goto Statement Considered Harmful"

Said *goto* is too primitive and an invitation to make a mess of one's programs

Advocated structured programming

Spaghetti Code

A BASIC Example

```
10      GOTO 40
20      STOP
30      GOTO 60
40      IF N < 1 THEN GOTO 20
50      J = 1
60      GOTO 70
70      PRINT J
80      GOTO 110
90      IF J>N THEN GOTO 20
100     GOTO 30
110     J = J+1
120     GOTO 90
```

D-Structures (Dijkstra)

A class of simple control structures:

- Basic actions (assignment, subroutine call, …)
- Selection (`if-then-else`)
- Iteration (`while`)
- Sequence of D-structures

Boehm – Jacopini Theorem (1966)

1) Any proper program can be written using only D-Structures

2) For any proper program, there exists a functionally equivalent program which uses only D-structures

- **Proper program** – One-entry, one-exit, no infinite loops, no unreachable code
- **Functionally equivalent programs** – Given the same input, they produce the same output

Ada has a goto!

```
loop
   get( x );
   if x = 0 then
     goto Finished;
   end if;
   sum := sum + x;
end loop;
<<Finished>>
put( sum );
```

Restricted GOTOs

FORTRAN – GOTO restricted to current subprogram

Pascal – GOTO cannot jump into a block, loop, or if-then-else from outside

Ada – Similar to Pascal, and has EXIT, RETURN

Java – Gone! (has `exit`, `break` & `return`)

C# - Still there!

Selection Statements

Components

- A control expression
- Statement(s) selected by the control

Common forms include

- IF-THEN
- IF-THEN-ELSE
- CASE

Implemented as conditional branches in machine/assembler

The IF Statement

FORTRAN's first generation IF:

```
      IF (X.GT.0) X = X – 1

      IF (X.LE.0) GO TO 20
         X = X - 1
   20 CONTINUE
```

The IF Statement

ALGOL60 added an ELSE clause

Both the IF-THEN and IF-THEN-ELSE support two-way selection

Perl's `unless` reverses the logic of the selection:

```
unless ($x == 0) {
   $z = $y/$x;
}
```

Or Ruby:

```
z = y / x unless x == 0
```

The IF Statement

Ada, has *elsif* clauses to support multi-way selection:
(so does Perl and Ruby)

```
if (x < 10) then
   …
elsif (x < 20) then
   …
elsif (x < 30) then
   …
else
   …
endif;
```

Multi-way Selection

Three way selection - FORTRAN arithmetic IF:

```
      IF (2*X+1)  10, 20, 30
   10    …
      GO TO 40
   20    …
      GO TO 40
   30    …
   40  CONTINUE
```

< 0 **= 0** **> 0**

Multi-way Selection

FORTRAN Computed GOTO

```
            X = 1   2   3   4
      GO TO (10, 20, 30, 40), X
   10    …
      GO TO 50
   20    …
      GO TO 50
   30    …
      GO TO 50
   40    …
   50  CONTINUE
```

Does this remind you of anything?

Multi-way Selection

C++/Java Switch Statement

```
switch (x) {
   case 1: … ;
           break;
   case 2:
   case 3: … ;
           break;
   default: …;
}
```

C# `switch` Statement

Case "fall through" is not allowed, except when a case is empty

`goto` can be used to force fall through

```
switch (x) {
   case 1:
   case 2:   // ok to stack cases
       …;
      goto case 3;   // forced fall through
   case 3: … ;
      break;
   default: …;
}
```

The case selector expression can also be a string

Multi-way Selection

Ada CASE Statement

```
CASE X IS
   WHEN 1..5 | 9  => …;
   WHEN 6, 7      => …;
   WHEN 8         => …;
   WHEN OTHERS    => …;
END CASE;
```

Iterative Statements

Components

- Type of control (counter, logic)
- Location of control mechanism (pretest, postest, …)
- Number of exits allowed

Enumeration (Counter) Controlled Loops

Uses a loop control variable to determine the number of iterations

Often the simplest loop to use, but…

Have the most complex design

Enumeration Controlled Loops

FORTRAN DO loop

```
      DO 20 I = 1, 10, 1
        …
   20 CONTINUE
```

Lower limit, Upper limit, Step (optional)

The LCV is updated automatically

Originally it was a post test loop (condition was tested at end of loop)

FORTRAN77 changed this to pre test

Enumeration Controlled Loops

"Operational semantics" of original DO loop:

```
      Lower = 1
      Upper = 10
      Step = 1
      I = Lower
L1:     -- body of loop
      I = I + Step
      IF I <= Upper GO TO L1
```

Enumeration Controlled Loops

Operational semantics of FORTRAN77 DO loop (simplified) :

```
      Lower = 1
      Upper = 10
      Step = 1
      I = Lower
L1:   IF I > Upper GO TO L2
          -- body of loop
      I = I + Step
      GO TO L1
L2:
```

Enumeration Controlled Loops

Pascal FOR loop example:

```
var i  : Integer;
for i := 1 to 10 do
  begin
    …
  end
```

Loop control variable semantics

- Bounds are evaluated once on entry to loop
- Cannot be changed in the loop
- Cannot be a parameter (must be a local var)
- Cannot be passed by reference to a subroutine
- Is undefined on loop exit

Enumeration Controlled Loops

C/Java FOR loop

```
for( int i = 0; i <= 10; i++) {
   …
}
```

Initialize; Pretest condition; Update *after* iteration

The for loop does not have to be used as a counting loop (C++ examples):

```
for (cin >> i; i != 0; cin >> i) { … }
for (p = head; p != null; p = p->next) { … }
```

Logic Controlled Loops

More general than counter controlled

Pretest loops

≥ 0 iterations

while loops

Posttest loops

≥ 1 iterations

Repeat-until, do-while loops

General loops

Programmer selected exit points

Logic Controlled Loops

Pretest loop (C++):

```
sum = 0;
cin >> val;
while (val >= 0) {
  sum = sum + val;
  cin >> val;
}
```

Operational Semantics:

```
    sum = 0
    read(val)
L1: if val<0 goto L2
    sum = sum + val
    read(val)
    goto L1
L2:
```

Logic Controlled Loops

Posttest loop:

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

Operational Semantics:

```
    digits = 0
    read( val)
L1: val = val / 10
    digits = digits + 1
    if val > 0 then goto L1
```

Logic Controlled Loops

General loop:

```
sum = 0;
while (true) {
  cin >> val;
  if (val==0) break;
  sum = sum + val;
}
```

Operational Semantics:

```
    sum = 0
L1: read( val)
    if val = 0 then goto L2
    sum = sum + val
    goto L1
L2:
```

Ada Loops

EBNF syntax

<Ada-loop> ::= [<iteration-spec>] **loop**
<loop-body>
end loop
<iteration-spec> ::= **while** <condition> |
for <index-param> **in** [**reverse**] <range>

Note: The <index-param> of a **for** loop has the loop body as its scope

Data Controlled Loop

Perl's foreach loop

```
@a = (10, 25, 51, 68, 50, 32);
foreach $x (@a) {
   print $x;
}
```

Can also use

```
@a = (10, 25, 51, 68, 50, 32);
foreach (@a) {
   print;
}
```

Data Controlled Loop

C# has a similar construct

```
int[] a = new int []{10,25,51,68,50,32};
foreach (int x in a) {
   Console.Write(x);
}
```

Java for loop version for classes with iterators

```
int [] a = {10,25,51,68,50,32};
for (int x : a) {
   System.out.print(x)
}
```

Looping in Eiffel

Incorporates initialization code with the loop:

```
from
  i := 1;
until i = 10 loop
  …
  i := i + 1;
end;
```

```
from
  read x;
until x = 0 loop
  …
  read x;
end;
```

Looping in Eiffel

Linear search

```
from
  i := 0;
until i >= A.Size or A[i] = key
loop
  i := i + 1;
end;
```

Guarded Commands
(Dijkstra, 1975)

The guarded if (non-deterministic)

- Conditions are called *guards*
- Evaluate **all** guards
- Statements with TRUE guards are *open*
- Randomly choose one of the open guards and execute the statement

```
if g1 -> stmt1;
[] g2 -> stmt2;
 …
[] gk -> stmtk;
fi
```

Guarded Commands

Example to find maximum of 2 values

```
if x >= y -> max := x;
[] y >= x -> max := y;
fi
```

Note that the case where x = y is non-deterministically handled by either statement

Guarded Commands

Guarded do loop

- Evaluates all guards
- Non-deterministically select an open statement to execute
- Exit when all guards are false

```
do g1 -> stmt1;
[] g2 -> stmt2;
 …
[] gk -> stmtk;
od
```

Guarded Commands

Example to sort 4 values in ascending order so that $w \le x \le y \le z$

```
do w > x -> swap(w, x);
[] x > y -> swap(x, y);
[] y > z -> swap(y, z);
od
```

Guarded Commands

Guarded commands are important in concurrent programming