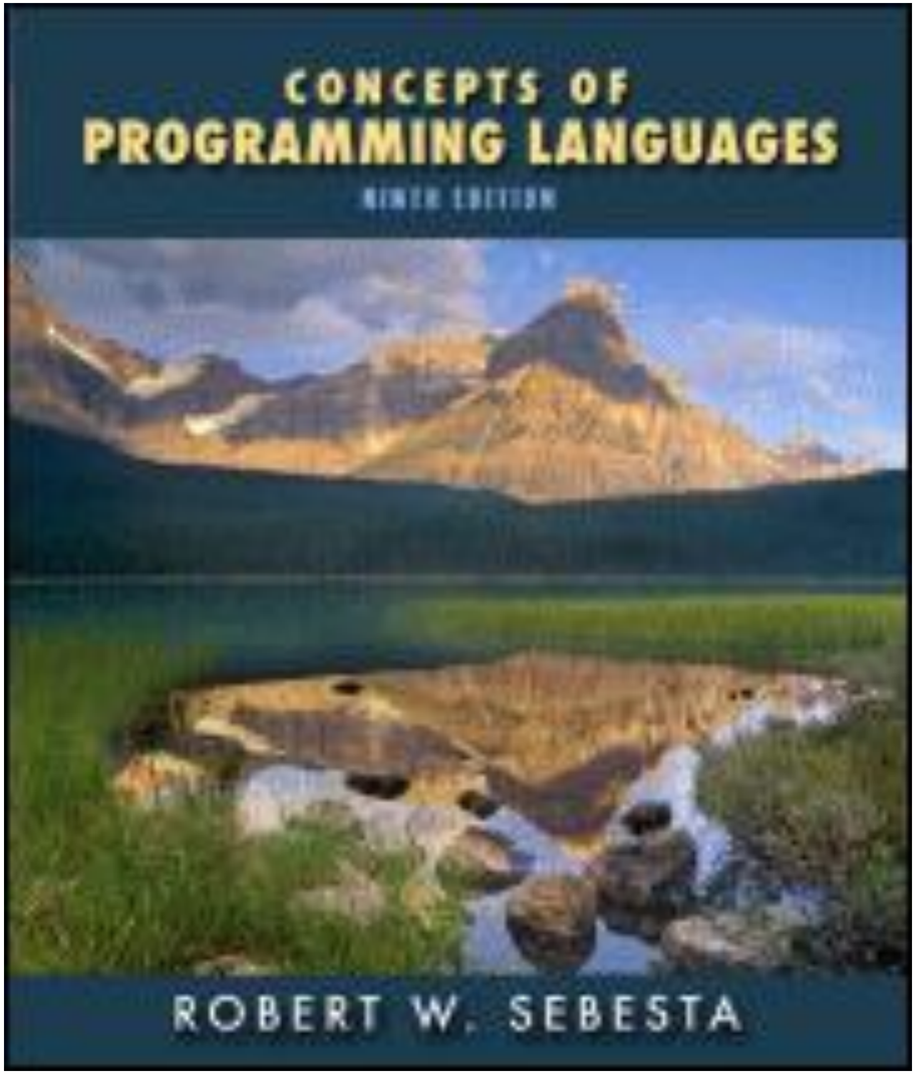
Chapter 8

Statement-Level Control Structures



Ch08 – Statement-Level Control Structures

8.1 Introduction*

8.2 Selection Statements

8.3 Iteration Statements

8.4 Unconditional Branching*

8.5 Guarded Commands*

8.6 Conclusions*

8.2 Selection Statements

- Implementing Multiple Selection Structures (8.2.2.3)
  - How to implement the following switch statement?

```
switch (exp) {
case 1: S1; break;
case 2: S2; break;
case 3: case 4: S34; break;
default: Sd;
}
```

- Method 1: Sequence of if’s

```
v=exp;
if (v==1) S1;
else if (v==2) S2;
else if (v==3 || v==4) S34;
else Sd;
```

Property

The latter cases take a longer time to reach.

Acceptable only when the number of cases is small.

8.2 Selection Statements

- Method 2 – Array of labels

```
label dispatch[5]={c1,c2,c34,c34,d};
v=exp;
i=1<=v&&v<=4? v-1: 4;
goto dispatch[i];
c1: S1; goto exit;
c2: S2; goto exit;
c34: S34; goto exit;
d: Sd;
exit:;
```

8.2 Selection Statements

- Method 2 (Cont’d)

  The following perl program simulates the preceding code by representing each label as a function:

```
# This program uses an array of references to functions.
sub c1 { print "S1\n"; }   # arbitrary action
sub c2 { print "S2\n"; }
sub c34 { print "S34\n"; }
sub d { print "Sd\n"; }
@dispatch=(\&c1,\&c2,\&c34,\&c34,\&d);
$v=3;                        # let exp=3, say
$i=1<=$v&&$v<=4? $v-1: 4;
&{$dispatch[$i]};    # passing @_ (=()); or, &{$dispatch[$i]}();
```

8.2 Selection Statements

- Method 2 (Cont’d)
  - On &

    To call a subroutine foo directly, we may write

```
&foo(args)     // full syntax
foo(args)      // & is optional with parentheses
foo args       // () is optional, if sub predeclared
&foo           // pass current @_ to foo
               // &foo args   is ill-formed
```

Example

```
sub foo { print @_; }   // 123
@_=(1,2,3);             // global @_=() initially
&foo;                   // foo; foo();  ⇐ both pass ()
```

8.2 Selection Statements

- On & (Cont'd)

  To create a reference to subroutine foo and call it, write

```
$ref=\&foo;            # & isn't optional here
&$ref(args)            # & isn't optional here
$ref->(args)           # unless using infix notation
&$ref                  # pass current @_
                  # &$ref args  is ill-formed
```

- On {}

```
&{$ref}(args) # {} is optional here
&{$ref}                # {} is optional here
&{$dispatch[$i]}       # {} is necessary here
                       # &$dispatch[$i];  is ill-formed
```

8.2 Selection Statements

- Method 2 (Cont’d)

  Method 2 isn’t good when the range of case values is large or the array index is hard to compute, e.g.

  ```
  switch (exp) {
  case 7: S7; break;
  case 911: S911; break;
  case 32767: S32767; break;
  default: Sd;
  }
  Try1:  label dispatch[32768]={d…,c7,d… ,c911,d… ,c32767};
  Try2:  label dispatch[4]={c7,c911,c32767,d};
         v=exp; i=v==7? 0: v==911? 1: v==32767? 2: 3;
  ```

8.2 Selection Statements

- Method 3 – Hash table

  For the preceding example, build a hash table containing (case-value, label) pairs (7,c7), (911,c911), (32767,c32767)

```
# Perl simulation of label as function
sub c7 { print "S7\n"; }
sub c911 { print "S911\n"; }
sub c32767 { print "S32767\n"; }
sub d { print "Sd\n"; }
%dispatch=(7=>\&c7,911=>\&c911,32767=>\&c32767);
$v=911;            # let exp=911, say
if (exists $dispatch{$v}) { &{$dispatch{$v}}; }  # passing @_
else { d; }
```

8.3 Iteration Statements

- Counter-Controlled Loops (8.3.1)
  - Loop variable

    Loop parameters: Initial, Terminal, Stepsize
  - Q: Evaluate loop parameters once or every iteration?
    - Once – Fortran

```
n=5; s=2          !      ini=1; step=s
do i=1,n,s        !      count=max(int(n-ini+step)/step,0)
    n=n+1         !      i=ini
    s=s+1         ! 10  If (count==0) goto 20
    print *,i,n,s !      <loop body>
end do            !      i=i+step; count=count-1; goto 10
                  ! 20
```

8.3 Iteration Statements

- Every iteration – C-based loop

```
n=5; s=2;
for (i=1;i<=n;i+=s) {
    n++; s++;
    cout << i << n << s;        // 163   474
}
```

○ Q: Can the loop variable be modified inside loop body?

- No – Fortran, Ada, Pascal
- Yes – C-based loop

```
for ($i=1;$i<=5;$i++) { print $i; $i++; }    // 135
```

N.B. Recall that

```
for $i (1..5) { print $i; $i++; }             // 12345
```

8.3 Iteration Statements

- Q: What is the scope of the loop variable?
  - Invisible outside the loop

    Ada

    C-based loop with locally declared loop variable

    e.g. for (int i=1;i<=5;i++) { ... }
  - Visible outside the loop

    C-based loop with nonloally declared loop variable

    e.g. int i; for (i=1;i<=5;i++) { ... }

    Fortran

    (the loop variable has its most recently assigned value)

    Pascal (the loop variable is undefined)

8.3 Iteration Statements

- Comment on the scope of Perl’s loop variable

```
$i=0;
sub sub2 { print $i; }
sub1();
print $i;
# lexical scope
sub sub1 { for ($i=1;$i<=5;$i++) { sub2; } }            #123456
sub sub1 { for (my $i=1;$i<=5;$i++) { sub2; } }         #000000
sub sub1 { for my $i (1..5) { sub2; } }                 #000000
# "unusual" dynamic scope
sub sub1 { for (local $i=1;$i<=5;$i++) { sub2; } }      #123450
sub sub1 { for $i (1..5) { sub2; } }                    #123450
```