MODULE 3

Control structures

- Statements that provide some means of selecting among alternative control flow paths (of statement execution) and some means of causing the repeated execution of statements or sequences of statements are called **control statements.**
- A **control structure** is a control statement and the collection of statements whose execution it controls.

Selection Statements

- A **selection statement** provides the means of choosing between two or more execution paths in a program.
- Selection statements fall into two general categories: two-way and n-way, or multiple selection.

Two-Way Selection Statements

- The general form of a two-way selector is as follows:

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

Design Issues

- The design issues for two-way selectors can be
- What is the form and type of the expression that controls the selection?
- How are the then and else clauses specified?
- How should the meaning of nested selectors be specified?

The Control Expression

- Control expressions are specified in parentheses if the **`then`** reserved word (or some other syntactic marker) is not used to introduce the then clause.
- In C89, which did not have a Boolean data type, arithmetic expressions were used as control expressions.
- In other contemporary languages, only Boolean expressions can be used for control expressions.

Clause Form

- In many contemporary languages, the then and else clauses appear as either single statements or compound statements. One variation of this is Perl, in which all then and else clauses must be compound statements, even if they contain single statements.
- Many languages use braces to form compound statements, which serve as the bodies of then and else clauses.
- In Fortran 95, Ada, Python, and Ruby, the then and else clauses are statement sequences, rather than compound statements

- Python uses indentation to specify compound statements. For example,

```
if x > y :
x = y
print "case 1“
```

Nesting Selectors

- That ambiguous grammar was as follows:

<if_stmt> **if** <logic_expr> **then** <stmt> | **if** <logic_expr> **then** <stmt> **else** <stmt>

- The issue was that when a selection statement is nested in the then clause of a
- selection statement, it is not clear to which if an else clause should be associated.
- Consider Java-like code:

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

- This statement can be interpreted in two different ways, depending on whether the else clause is matched with the first then clause or the second. Notice that the indentation seems to indicate that the else clause belongs with the first then clause.

- To force the alternative semantics in Java, the inner **`if`** is put in a compound, as in

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

- C, C++, and C# have the same problem as Java with selection statement nesting. Because Perl requires that all then and else clauses be compound, it does not. In Perl, the previous code would be written as

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

- If the alternative semantics were needed, it would be

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

- in Ruby as follows:

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

- Because the **`end`** reserved word closes the nested **`if`**, it is clear that the else clause is matched to the inner then clause.
- in Python, statement above:

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

- If the line **`else:`** were indented to begin in the same column as the nested **`if`**, the else clause would be matched with the inner **`if`**.

Selector Expressions

- In the functional languages ML, F#, and LISP, the selector is not a statement; it is an expression that results in a value. Therefore, it can appear anywhere any other expression can appear. Consider the following example selector written in F#:

```
let y =
    if x > 0 then x
    else 2 * x;;
```

- This creates the name `y` and sets it to either `x` or `2 * x`, depending on whether `x` is greater than zero.

Multiple-Selection Statements

- The **multiple-selection** statement allows the selection of one of any number of statements or statement groups.

Design Issues

- What is the form and type of the expression that controls the selection?
- How are the selectable segments specified?
- Is execution flow through the structure restricted to include just a single selectable segment?
- How are the case values specified?
- How should unrepresented selector expression values be handled, if at all?

Examples of Multiple Selectors

- The C multiple-selector statement, **`switch`**, which is also part of C++, Java,and JavaScript, is a relatively primitive design. Its general form is

```
switch (expression) {
case constant_expression1:statement1;
. . .
case constantn: statement_n;
[default: statementn+1]
}
```

- where the control expression and the constant expressions are some discrete type.
- The optional **`default`** segment is for unrepresented values of the control expression. The **`switch`** statement does not provide implicit branches at the end of its code segments. This allows control to flow through more than one selectable code segment on a single execution.
- Consider the following example:

```
switch (index) {
case 1:
```

```
case 3: odd += 1;
sumodd += index;
case 2:
case 4: even += 1;
sumeven += index;
default: printf("Error in switch, index = %d\n", index);
}
```

- This code prints the error message on every execution.
- The **`break`** statement, which is actually a restricted goto, is normally used for exiting **`switch`** statements.
- The following **`switch`** statement uses **`break`** to restrict each execution to a single selectable segment:

```
switch (index) {
case 1:
case 3: odd += 1;
sumodd += index;
break;
case 2:
case 4: even += 1;
sumeven += index;
break;
default: printf("Error in switch, index = %d\n", index);
}
```

- The C switch statement has virtually no restrictions on the placement of the case expressions, which are treated as if they were normal statement labels.

```
switch (x)
default:
if (prime(x))
case 2: case 3: case 5: case 7:
process_prime(x);
else
case 4: case 6: case 8: case 9: case 10:
process_composite(x);
```

- The C# switch statement differs from that of its C-based predecessors in two ways. First, C# has a static semantics rule that disallows the implicit execution of more than one segment. The rule is that every selectable segment must end with an explicit unconditional branch statement: either a **`break`**, which transfers control out of the **`switch`** statement, or a goto, which can transfer control to one of the selectable segments

For example,

```
switch (value) {
   case -1:
      Negatives++;
      break;
   case 0:
      Zeros++;
      goto case 1;
   case 1:
      Positives++;
   default:
      Console.WriteLine("Error in switch \n");
}
```

Ruby's case expressic

```
case
when Boolean_expression then expression
...
when Boolean_expression then expression
[else expression]
end
```

The semantics of this case expression is that the Boolean expressions are evaluated one at a time, top to bottom. The value of the case expression is the value of the first then expression whose Boolean expression is true. The else represents true in this statement, and the else clause is optional. For example,[4]

```
leap = case
       when year % 400 == 0 then true
       when year % 100 == 0 then false
       else year % 4 == 0
       end
```

Implementing Multiple Selection Structures

- Implementing such a statement must be done with multiple conditional branch instructions.

```
switch (expression) {
 case constant_expression1: statement1;
  break;
 . . .
 case constantn: statementn;
  break;
 [default: statementn+1]
}
```

One simple translation of this statement follows:

```
Code to evaluate expression into t
goto branches
label1: code for statement1
            goto out
. . .
labeln: code for statementn
            goto out
default: code for statementn+1
            goto out
branches: if t = constant_expression1 goto label1
             . . .
             if t = constant_expressionn goto labeln
             goto default
out:
```

Multiple Selection Using `if`

- To alleviate the poor readability of deeply nested two-way selectors, some languages, such as Perl and Python, have been extended specifically for this use.
- The extension allows some of the special words to be left out. In particular, else-if sequences are replaced with a single special word, and the closing special word on the nested `if` is dropped.
- The nested selector is then called an **else-if clause**.
- Consider the following Python selector statement

```
if count < 10 :
  bag1 = True
elif count < 100  :
  bag2 = True
elif count < 1000 :
  bag3 = True
```

which is equivalent to the following:

```
if count < 10 :
  bag1 = True
else :
  if count < 100 :
    bag2 = True
  else :
    if count < 1000 :
      bag3 = True
    else :
      bag4 = True
```

- The Python example if-then-else-if statement above can be written as the Ruby **`case`** statement:

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

The Scheme multiple selector, which is based on mathematical conditional expressions, is a special form function named `COND`. `COND` does not require a fixed number of actual parameters. Each parameter to `COND` is a pair of expressions in which the first is a predicate (it evaluates to either `#T` or `#F`).

The general form of COND is

```
(COND
  (predicate1 expression1)
  (predicate2 expression2)
  ...
  (predicaten expressionn)
  [(ELSE expressionn+1)]
)
```

where the ELSE clause is optional.

Consider the following example call to COND:

```
(COND
  ((> x y) "x is greater than y")
  ((< x y) "y is greater than x")
  (ELSE "x and y are equal")
)
```

Note that string literals evaluate to themselves, so that when this call to COND is evaluated, it produces a string result.

Iterative Statements

- An **iterative statement** is one that causes a statement or collection of statements to be executed zero, one, or more times. An iterative statement is often called a **loop**.
- two basic design questions:
  - How is the iteration controlled?
  - Where should the control mechanism appear in the loop statement?
- The **body** of an iterative statement is the collection of statements whose execution is controlled by the iteration statement.
- **pretest** means that the test for loop completion occurs before the loop body is executed and **posttest** to mean that it occurs after the loop body is executed. The iteration statement and the associated loop body together form an **iteration statement.**

Counter-Controlled Loops

- A counting iterative control statement has a variable, called the **loop variable**, in which the count value is maintained. It also includes some means of specifying the **initial** and **terminal** values of the loop variable, and the difference between sequential loop variable values, often called the **stepsize**.
- The initial, terminal, and stepsize specifications of a loop are called the **loop parameters**.

Design Issues

- What are the type and scope of the loop variable?
- 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 loop parameters be evaluated only once, or once for every iteration?

The `for` Statement of the C-Based Languages

- The general form of C's `for` statement is

  `for` (expression_1; expression_2; expression_3)

  loop body

- The loop body can be a single statement, a compound statement, or a null statement.
- The first expression is for initialization and is evaluated only once, when the `for` statement execution begins. The second expression is the loop control and is evaluated before each execution of the loop body. As is usual in C, a zero value means false and all nonzero values mean true. Therefore, if the value of the second expression is zero, the `for` is terminated; otherwise, the loop body statements are executed.

for statement is shown next. Because C expressions can be used as statements, expression evaluations are shown as statements.

```
     expression_1
loop:
     if expression_2 = 0 goto out
     [loop body]
     expression_3
     goto loop
out: . . .
```

Following is an example of a skeletal C **for** statement:

```
for (count = 1; count <= 10; count++)
  . . .
}
```

All of the expressions of C's **for** are optional. An absent second expression is considered true, so a **for** without one is potentially an infinite loop.When multiple expressions are used in a single expression of a **for** statement, they are separated by commas. The scope of a variable defined in the **for** statement is from its definition to the end of the loop body.

```
for (count1 = 0, count2 = 1.0;
     count1 <= 10 && count2 <= 100.0;
     sum = ++count1 + count2, count2 *= 2.5);
```

The operational semantics description of this is

```
     count1 = 0
     count2 = 1.0
loop:
     if count1 > 10 goto out
     if count2 > 100.0 goto out
     count1 = count1 + 1
     sum = count1 + count2
     count2 = count2 * 2.5
     goto loop
```

The `for` Statement of Python

- The general form of Python's **`for`** is

```
for loop_variable in object:
  - loop body
[else:
  - else clause]
```

The loop variable is assigned the value in the object, which is often a range, one for each execution of the loop body. The else clause, when present, is executed if the loop terminates normally.

Consider the following example:

```
for count in [2, 4, 6]:
  print count
```

produces

```
2
4
6
```

For most simple counting loops in Python, the **`range`** function is used. **`range`** takes one, two, or three parameters. The following examples demonstrate the actions of **`range`**:

```
range(5) returns [0, 1, 2, 3, 4]
range(2, 7) returns [2, 3, 4, 5, 6]
range(0, 8, 2) returns [0, 2, 4, 6]
```

Note that **`range`** never returns the highest value in a given parameter range.

Logically Controlled Loops

- collections of statements must be repeatedly executed, but the repetition control is based on a Boolean expression rather than a counter.

Design Issues

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

Examples

- The C-based programming languages include both pretest and posttest logically controlled loops.
- The pretest and posttest logical loops have the following forms:

```
while (control_expression)
    loop body
```

and

```
do
    loop body
while (control_expression);
```

These two statement forms are exemplified by the following C# code segments:

```
sum = 0;
indat = Int32.Parse(Console.ReadLine());
while (indat >= 0) {
  sum += indat;
  indat = Int32.Parse(Console.ReadLine());
}

value = Int32.Parse(Console.ReadLine());
do {
  value /= 10;
  digits ++;
} while (value > 0);
```

- In the pretest version of a logical loop (**`while`**), the statement or statement segment is executed as long as the expression evaluates to true. In the posttest version (**`do`**), the loop body is executed until the expression evaluates to false.

can be compound. The operational semantics descriptions of those two statements follows:

`while`

```
loop:
  if control_expression is false goto out
  [loop body]
  goto loop
out: . . .
```

`do-while`

```
loop:
  [loop body]
  if control_expression is true goto loop
```

User-Located Loop Control Mechanisms

- The design issues for such a mechanism are the following:
  - Should the conditional mechanism be an integral part of the exit?
  - Should only one loop body be exited, or can enclosing loops also be exited?
- C, C++, Python, Ruby, and C# have unconditional unlabeled exits (**`break`**).
- Java and Perl have unconditional labeled exits (**`break`** in Java, **`last`** in Perl).
- C, C++, and Python include an unlabeled control statement, **`continue`**, that transfers control to the control mechanism of the smallest enclosing loop. Both **`last`** and **`break`** provide for multiple exits from loops, which may seem to be somewhat of a hindrance to readability.

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

A negative value causes the assignment statement to be skipped, and control is transferred instead to the conditional at the top of the loop. On the other hand, in

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

a negative value terminates the loop.

Unconditional Branching

- An **unconditional branch statement** transfers execution control to a specified location in the program.
- The unconditional branch, or goto, is the most powerful statement for controlling the flow of execution of a program's statements.
- The goto has stunning power and great flexibility .
- goto statements can make programs very difficult to read, and as a result, highly unreliable and costly to maintain.
- A few languages have been designed without a goto—for example, Java, Python, and Ruby.
- The relatively new language, C#, includes a goto, even though one of the languages on which it is based, Java, does not. One legitimate use of C#'s goto is in the **`switch`** statement.

Guarded Commands

- New and quite different forms of selection and loop structures were suggested by Dijkstra.
- His primary motivation was to provide control statements that would support a program design methodology that ensured correctness during development rather than when verifying or testing completed programs.
- Another motivation for developing guarded commands is that nondeterminism is sometimes needed in concurrent programs.
- Another motivation is the increased clarity in reasoning.
- a selectable segment of a selection statement in a guarded-command statement can be considered independently of any other part of the statement.

Dijkstra's selection statement has the form

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

- The closing reserved word, **`fi`**, is the opening reserved word spelled backward.
- This form of closing reserved word is taken from ALGOL 68.
- The small blocks, called *fatbars*, are used to separate the guarded clauses and allow the clauses to be statement sequences.
- Each line in the selection statement, consisting of a Boolean expression (a guard) and a statement or statement sequence, is called a **guarded command**.
- All of the Boolean expressions are evaluated each time the statement is reached during execution. If more than one expression is true, one of the corresponding statements can be nondeterministically chosen for execution.
- An implementation may always choose the statement associated with the first Boolean expression that evaluates to true. But it may choose any statement associated with a true Boolean expression.
- If none of the Boolean expressions is true, a run-time error occurs that causes program termination.

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

If `i = 0` and `j > i`, this statement chooses nondeterministically between the first and third assignment statements. If `i` is equal to `j` and is not zero, a run-time error occurs because none of the conditions is true.

- The loop structure proposed by Dijkstra has the form

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

- The semantics of this statement is that all Boolean expressions are evaluated on each iteration. If more than one is true, one of the associated statements is nondeterministically (perhaps randomly) chosen for execution, after which the expressions are again evaluated. When all expressions are simultaneously false, the loop terminates.
- Consider the following problem: Given four integer variables, `q1`, `q2`, `q3`, and `q4`, rearrange the values of the four so that `q1 ≤q2 ≤ q3 ≤q4`.
- uses guarded commands to solve the same problem but in a more concise and elegant way

```
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
```

- Although they have not been adopted as the control statements of a language, part of the semantics appear in the concurrency mechanisms of CSP and Ada and the function definitions of Haskell.

SUBPROGRAMS-INTRODUCTION

- Subprograms are the fundamental building blocks of programs and are therefore among the most important concepts in programming language design
- Two fundamental abstraction facilities can be included in a programming language: process abstraction and data abstraction
- Process abstraction, in the form of subprograms, has been a central concept in all programming languages.

Fundamentals of Subprograms

General Subprogram Characteristics

- Each subprogram has a single entry point.

• The calling program unit is suspended during the execution of the called subprogram, which implies that there is only one subprogram in execution at any given time.

• Control always returns to the caller when the subprogram execution terminates.

Basic Definitions

- A **subprogram definition** describes the interface to and the actions of the subprogram abstraction. A **subprogram call** is the explicit request that a specific subprogram be executed.
- A subprogram is said to be **active** if, after having been called, it has begun execution but has not yet completed that execution.
- A **subprogram header**, which is the first part of the definition, serves several purposes. First, it specifies that the following syntactic unit is a subprogram definition of some particular kind.
- In languages that have more than one kind of subprogram, the kind of the subprogram is usually specified with a special word. Second, if the subprogram is not anonymous, the header provides a name for the subprogram. Third, it may optionally specify a list of parameters.
- Consider the following header examples:

```
def adder(parameters):
```

- This is the header of a Python subprogram named `adder`. Ruby subprogram headers also begin with **`def`**. The header of a JavaScript subprogram begins with **`function`**.
- In C, the header of a function named **`adder`** might be as follows:

```
void adder (parameters)
```

- The reserved word **`void`** in this header indicates that the subprogram does not return a value.

- All Lua functions are anonymous, although they can be defined using syntax that makes it appear as though they have names. For example, consider the following identical definitions of the function `cube`:

```
function cube(x) return x * x * x end

cube = function (x) return x * x * x end
```

- The first of these uses conventional syntax, while the form of the second more accurately illustrates the namelessness of functions.
- The **parameter profile** of a subprogram contains the number, order, and types of its formal parameters. The **protocol** of a subprogram is its parameter profile plus, if it is a function, its return type.
- Subprograms can have declarations as well as definitions. This form parallels the variable declarations and definitions in C, in which the declarations can be used to provide type information but not to define variables.
- declarations are often placed in header files. In most other languages (other than C and C++), subprograms do not need declarations, because there is no requirement that subprograms be defined before they are called .

Parameters

- The parameters in the subprogram header are called **formal parameters**.
- Subprogram call statements must include the name of the subprogram and a list of parameters to be bound to the formal parameters of the subprogram. These parameters are called **actual parameters.**