MODULE 3
Statement-Level Control Structures

Statement-Level Control Structures

- Introduction
- Selection Statements
- Iterative Statements
- Unconditional Branching
- Guarded Commands

Introduction

- Two additional linguistic mechanisms are necessary to make the computations in programs flexible and powerful:
  - 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.

  Statements that provide these kinds of capabilities are called **control statements.**
- Levels of Control Flow:

  1. Within expressions
  2. Among program units
  3. Among program statements

- Evolution:

  FORTRAN I control statements were based directly on IBM 704 hardware

- 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.
- Such statements are fundamental and essential parts of all programming languages
- 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 summarized as follows:

- 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 is not used to introduce the then clause.
- In those cases where the **then** reserved word (or alternative marker) is used, there is less need for the parentheses.

Clause Form

- In many contemporary languages, the then and else clauses appear as either single statements or compound statements.
- Many languages use braces to form compound statements, which serve as the bodies of then and else clauses.

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

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

- All statements equally indented are included in the compound statement.
- Notice that rather than **then**, a colon is used to introduce the **then** clause in Python.

Nesting Selectors

- 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.
- This problem is reflected in the semantics of selection statements.
- Consider the following 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.

- In Java, as in many other imperative languages, the static semantics of the language specify that the else clause is always paired with the nearest previous unpaired then clause.
- A static semantics rule, rather than a syntactic entity, is used to provide the disambiguation.
- So, in the example, the else clause would be paired with the second then clause.
- 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;
}
```

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.
- 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**

- The following is a summary of these 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, andJavaScript, is a relatively primitive design.
- Its general form is

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

- The **break** statement, which is actually a restricted goto, is normally used for exiting **switch** statements.

- Example:

```
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);
}
```

Implementing Multiple Selection Structures

- A multiple selection statement is essentially an n-way branch to segments of code, where n is the number of selectable segments.
- Implementing such a statement must be done with multiple conditional branch instructions.

- Consider the general form of the C switch statement, with breaks:

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

- 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

- In many situations, a **switch** or **case** statement is inadequate for multiple selection.
- For example, when selections must be made on the basis of a Boolean expression rather than some ordinal type, nested two-way selectors can be used to simulate a multiple selector.
- 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 (note that else-if is spelled **elif** in Python):

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

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**.
- General design Issues for iteration control statements:

  1. How is iteration controlled?

  2. Where should the control mechanism appear in the loop 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

- The following is a summary of these 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 expressions in a **for** statement are often assignment statements.
- 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.
- 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.
- If the first and/or third expressions are absent, no assumptions are made. For example, if the first expression is absent, it simply means that no initialization takes place.

- C's **for** is more flexible than the counting loop statement of Ada, because each of the expressions can comprise multiple expressions, which in turn allow multiple loop variables that can be of any type.
- The scope of a variable defined in the **for** statement is from its definition to the end of the loop body.

➢Logically Controlled Loops

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

➢**Design Issues**

- Because they are much simpler than counter-controlled loops, logically controlled loops have fewer 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 that are not special forms of their counter-controlled iterative statements.
- The pretest and posttest logical loops have the following forms:

while (control_expression)

loop body

and

do

loop body

while (control_expression);

➢User-Located Loop Control Mechanisms

- In some situations, it is convenient for a programmer to choose a location for loop control other than the top or bottom of the loop body.
- A syntactic mechanism for user-located loop control can be relatively simple, so its design is not difficult.
- 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.

- For example, consider the following:

```
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.
- However, using the goto carelessly can lead to serious problems.
- Without restrictions on use, imposed by either language design or programming standards, goto statements can make programs very difficult to read, and as a result, highly unreliable and costly to maintain.

Guarded Commands

- Each line in the selection statement, consisting of a Boolean expression (a guard) and a statement or statement sequence, is called a **guarded command.**
- Purpose: to support a new programming methodology (verification during program development)

1 Selection:

if <boolean> -> <statement>

[] <boolean> -> <statement>

…

[] <boolean> -> <statement>

fi

- Semantics: when this construct is reached,
  - - Evaluate all boolean expressions
  - - If more than one are true, choose one nondeterministically
  - - If none are true, it is a runtime error

2. Loops

do <boolean> -> <statement>

[] <boolean> -> <statement>

…

[] <boolean> -> <statement>

od

- Semantics: For each iteration:
  - - Evaluate all boolean expressions
  - - If more than one are true, choose one non-deterministically; then start loop again
  - - If none are true, exit loop