Statement ‘Level’ {

[Control Structure]

< Ashley Mae Turla >
< John Daves Baguio >
< Lord Christian Carl Regacho >

}

Topics of Chapter 8;

8.1 {

[Introduction]

}

Evolution < /1 > {

< **Fortran**, were, in effect, designed by the architects of the IBM 704. Between the mid-1960s and mid-1970s there were talks and arguments on Control Statements>

}

Evolution < /2 > {

< **All Programming Languages represented by Flowcharts can be coded with only two control statements.** A result of this is the unconditional branch statement which is useful but nonessential >

}

Control Structure; {

‘A **Control Structure** is a control statement and the collection of statements whose execution it controls’

Selection Control Statements

- known as branching statements or conditional or decision-making statements

Iteration Control Statements

- cause statements to be executed zero or more times, subject to some loop-termination criteria

}

Control Structure; {

‘All **selection and iteration constructs** control the execution of code segments.

Selection Control Statements

- Example: if, switch, conditional operators, ternary

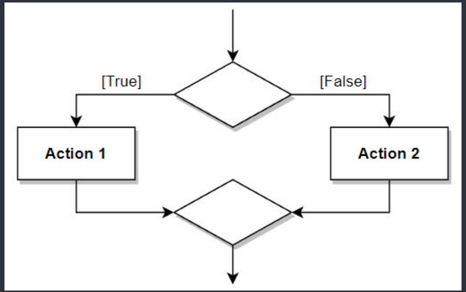
Iteration Control Statements

- Example: while-loop, do-while-loop, for-loop

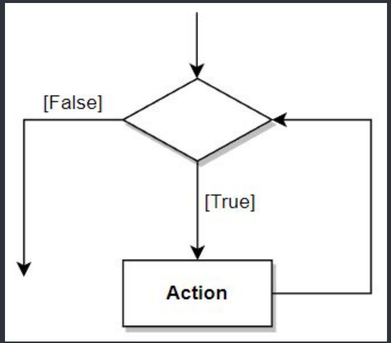
}

Question Time! {

Identify if the Diagram is a **Selection Statement** or an **Iteration Statement**.



Selection Statement



Iteration Statement

}

8.2 {

[Selection Statement]

}

Selection Statements; {

'A **Selection Statement** provides the means of choosing between two or more execution paths in a program.'

Two general categories:

- Two-way selectors
- Multiple-way selectors

}

Two-way Selection Statements {

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

Design Issues:

- What is the form and type of the control expression?
- 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, arithmetic expressions were used as control expressions. This can also be done in Python, C99, and C++.

In languages such as Ada, Java, Ruby, and C#, the control expression must be Boolean

}

The Control Expression; {

C

Parentheses Required

```c
if (a > b){
      printf(“A is Greater!”);
}
```

Python

Parentheses Optional

```python
if a > b:
      print(“A is Greater!”)
}
```

Java

Parentheses Required

```java
if (a > b)
      System.out.println(“A is Greater!”);
```

}

The Clause Form; {

'In many contemporary languages, the then and else clauses can be single statements or compound statements'

```
if (a>b)
    retVal = a;
else
    retVal = b;
```

Can this be Reduced to a Single Statement?

YES !!

C & Java

```
retVal = (a > b)? a : b;
```

Python

```
retVal = a if a > b else b;
```

(This is not ternary)

}

Nesting Selectors {

```
if_statement (condition)
    if_statement (condition)
        print statement;
else_statement
    print statement;
```

Question:
Which if-statement gets the else?

Answer:
The Nearest if-statement

The issue is that when a **selection statement** is **nested** in the then clause of a selection statement, it is not clear with which if an else clause should be associated.

}

Nesting Selectors {

To force an alternative semantics, compound statements may be used:

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

The above solution is used in C, Java, C++, and C#

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

The above solution is used in Ruby

}

Nesting Selectors {

- Perl

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

- Python

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

}

Multiple-Way 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?

}

Multiple-Way Selection: Examples; {

Switch (C, C++, and Java)

General form:

```
switch (expression) {
    case const_expr_1 : stmt_1;

    . . .

    case const_expr_n : stmt_n;
    [default : stmt_n+1]
}
```

}

Example; {

- Switch

```
switch (index) {
      case 1:
      case 3:  odd += 1;
               sumodd += index;
      case 2:
      case 4:  even += 1;
               sumeven += index;
      default: printf("Error");
}
```

We need to add **Breaks!**

}

Question!
What do you think this Code Prints?

Answer:
This code prints an Error Message on every Execution!

Likewise, the code for the 2 and 4 constants is executed every time the code at the 1 or 3 constants is executed.

Multiple-Way Selection: Examples; {

- Switch

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

The switch statement uses break to restrict each execution to a single selectable segment

}

Multiple-Way Selection: Examples; {

- Switch (C)

```
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);
```

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

}

Multiple-Way Selection: Examples; {

- Switch (**C#**)

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:

- Each selectable segment must end with an unconditional branch (goto or break)
- Also, in **C#** the control expression and the case constants can be strings

}

Multiple-Way Selection: Examples; {

- Switch (**C#**)

```
switch (value) {
    case -1:
        Negatives++;
        break;          // Unconditional Branch (break)
    case 0:
        Zeros++;
        goto case 1; // Unconditional Branch (goto)
    case 1:
        Positives++;
    default:
        Console.WriteLine("Error");  // Displays Strings
}
```

}

Multiple-Way Selection: Examples; {

- Switch (**Ada**)

```
case expression is
    when choice list ⇒ stmt_sequence;
    . . .
    when choice list ⇒ stmt_sequence;
    when others ⇒ stmt_sequence;
end case;
```

Ada design choices:

- Expressions can be ordinal type
- Segments can be single or compound
- Only one segment can be executed per execution of the construct
- Unpresented values are not allowed

}

Multiple-Way Selection: Examples; {

- Switch (**Ruby**)

```
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.

}

Multiple-Way Selection: Examples; {

- Switch (**Ruby**)

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

This case expression evaluates to true if year is a leap year.

The other Ruby case expression form is similar to the switch of Java. Perl and Python do not have multiple-selection statements.

}

Multiple-Way Selection using **if** {

‘**Multiple Selectors** can appear as direct extensions to two-way selectors, using else-if clauses,’

- **Python**

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

Equivalent to

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

}

Multiple-Way Selection using if {

- **Ruby**

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

}

8.3 { [Iterative Statements] }

Iterative Statements; {

- often called a loop
- repetitive execution of a statement or collection of statements
- accomplished either by iteration or recursion

Basic Design Questions:

1. How is the iteration controlled?
2. Where should the control mechanism appear in the loop statement?

}

Iterative Statements; {

Body

- collection of statements whose execution is controlled by the iteration statement

Iteration Statement

- iteration statement and the associated loop body together

Pretest

- test for loop completion occurs before the loop body

Posttest

- test for loop completion occurs after the loop body

}

Counter-Controlled Loops; {

- also known as definite repetition loop, since the number of iterations is known before the loop begins to execute
- repetition is managed by a loop variable

Loop Variable

- a variable that contains the count value
- includes some means of specifying the initial, terminal and stepsize values (Loop Parameters).

}

Counter-Controlled Loops; {

- more complex than logically controlled loops; their design is more demanding
- sometimes supported by machine instructions designed for that purpose
- Example: VAX computers have a very convenient instruction for the implementation of posttest counter-controlled loops, which Fortran had (mid-1970s)

}

Counter-Controlled Loops; {

Fortran

```
do count_variable = start,
stop [,step]
    <fortran statement(s)>
end do
```

```
!compute factorials
do n = 1, 10
    nfact = nfact*n
    !printing the value of n and
    its factorial
    print*, n, " ", nfact
end do
```

}

Counter-Controlled Loops; {

Design Issues:

1. What are the type and scope of the loop variable?
2. 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?
3. Should the loop parameters be evaluated only once, or once for every iteration?
4. What is the value of the loop variable after loop termination?

}

Counter-Controlled Loops; {

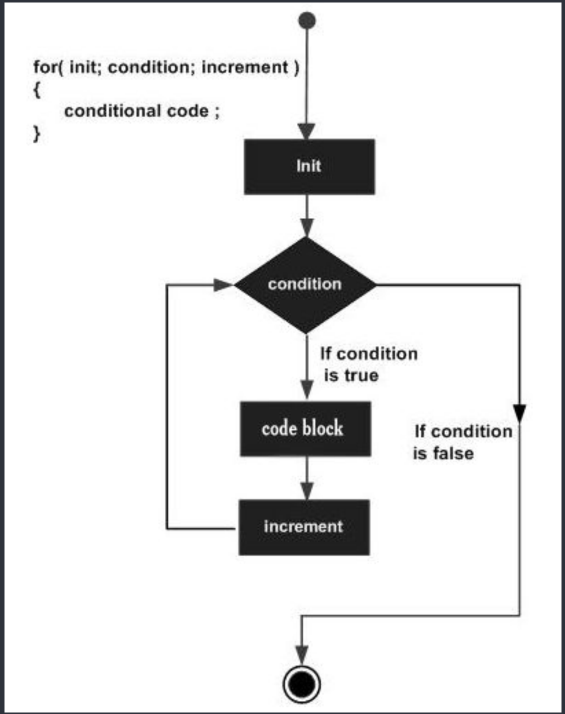
C-Based Loops

- Single Statement
- Compound Statement
- Null Statement

- one of the most flexible
- can easily model counting and logical loop structures

}

Counter-Controlled Loops; {



Java

```
public class Main{
    static void getSum(int num, int n)
    {
        int sum = 0;
        for (int i = 0; i < n; i++) {
            sum += num;
        }
        System.out.println(sum);
    }
}
```

}

Counter-Controlled Loops; {

C for Loop

- C's for is one of the most flexible
  - All of the expressions of C's for are optional
  - Note that C's for need not count. It can easily model counting and logical loop structures
  - There is no explicit loop variable and no loop parameters. All involved variables can be changed in the loop body

}

Counter-Controlled Loops; {

C for Loop

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

}

Counter-Controlled Loops; {

C-based languages

- C++ allows the control expression to be boolean
- C++ initial expression can include variable definitions
- Java and C# loop control expression is restricted to boolean

}

Counter-Controlled Loops; {

Python

- Loop variable is assigned the value in the object
- After loop termination, loop variable has the value last assigned to it

}

Counter-Controlled Loops; {

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

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

```
fruits = ["apple", "banana",
"cherry"]
for x in fruits:
  print(x)
```

}

Counter-Controlled Loops; {

- simple counting loops in Python use the range function

Range - takes one, two or three parameters

```
range(begin,end,step)
```

- 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]

- never returns the highest value in a given parameter range

}

Counter-Controlled Loops; {

```
for x in range(2, 30, 3):
    print(x)
```

Output

```
2
5
8
11
14
17
20
23
26
29
```

}

```
values = range(4)

for x in values:
    print(x)
```

Output

```
0
1
2
3
```

Counter-Controlled Loops; {

Functional Languages

- uses recursion rather than iteration

F#

```
let rec forLoop loopBody reps =
    if reps <= 0 then
        ()
    else
        loopBody()
        forLoop loopBody, (reps -
    1);;
```

}

Logically-Controlled Loops; {

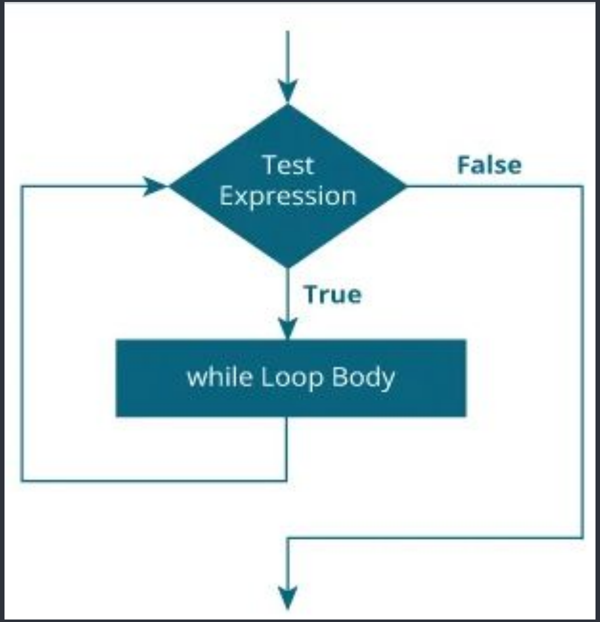
- more general than counter-controlled loops
- counting loop can be built with logical loop, but the reverse is not true
- repetition control is based on a Boolean expression rather than a counter

Design Issues

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

}

Logically-Controlled Loops; {



C-Based (Pretest)

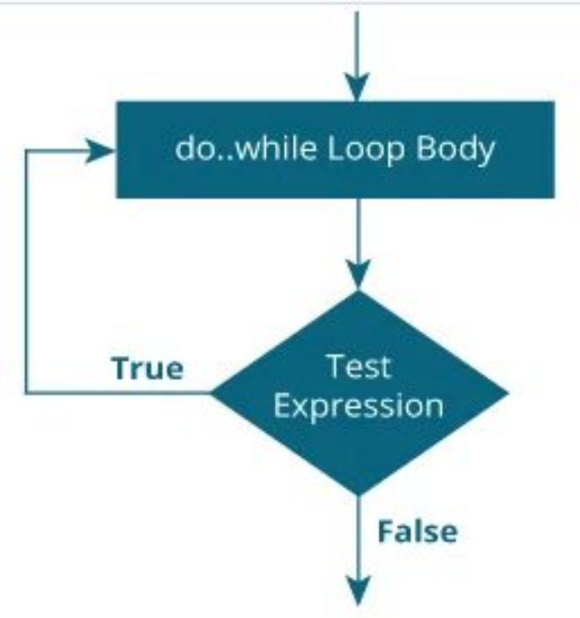
```
int main () {
   int a = 10;

   while(a < 20) {
      printf("value of a: %d\n", a);
      a++;
   }

   return 0;
}
```

}

Logically-Controlled Loops; {



C-Based (Posttest)

```
int main () {
    int a = 10;

    do {
        printf("value of a: %d\n", a);
        a=a+1;
    }while (a < 20);

   return 0;
}
```

}

Logically-Controlled Loops; {

C-Based Languages

C and C++

- It is legal to branch into both while and do loop bodies
- C89 version uses an arithmetic expression for control
- C99 and C++ may be either arithmetic or Boolean

Java

- control expression must be boolean type
- loop bodies cannot be entered anywhere except at their beginnings

}

Logically-Controlled Loops; {

Python

```
num = int(input("Enter a number: "))
fac = 1
i = 1
while i < num:
    fac = fac * i
    i = i + 1
print("Factorial of ", num, " is ", fac)
```

Output

```
Enter a number: 4
Factorial of  4  is  24
```

}

Logically-Controlled Loops; {

Python does not have a do-while loop, but we can emulate it

```
i = 1
while True:
    print(i)
    i = i + 1
    if(i > 3):
        break
```

```
secret_word = "python"
counter = 0

while True:
    word = input("Enter the secret word: ").lower()
    counter = counter + 1
    if word == secret_word:
        break
    if word != secret_word and counter > 7:
        break
```

}

Logically-Controlled Loops; {

Problem with Posttest

- infrequently useful
- somewhat dangerous; programmers sometimes forget that the loop body will always be executed at least once
- placing a posttest control physically after the loop body helps avoid such problems by making the logic clear

}

Logically-Controlled Loops; {

Functional Languages

- A pretest logical loop can be simulated in a purely functional form with a recursive function that is similar to the one used to simulate a counting loop

F#

```
let rec whileLoop test body =
    if test() then
        body()
        whileLoop test body
    else
        ();;
```

}

Logically-Controlled Loops; {

Other Languages

- Ada has a pretest version, but no posttest
- FORTRAN 95 has neither
- Perl and Ruby have two pretest logical loops, while and until. Perl also has two posttest loops

}

User-Located Loop Control Mechanisms; {

- choose a location for loop control other than top or bottom
- simple design for single loops
- fulfills a common need for goto statements using a highly restricted branch statement

Design Issues

1. Should the conditional mechanism be an integral part of the exit?
2. Should only one loop body be exited, or can enclosing loops also be exited?

}

User-Located Loop Control Mechanisms; {

Unconditional Statements

- control statements that do not need any condition to control the program execution flow
- continue, break, goto, return in C

Unconditional unlabelled exit

- only exits from the loop within which it is enclosed (innerloop)

Unconditional labelled exit

- exit out of a deeply nested set of loops (outerloop)

}

User-Located Loop Control Mechanisms; {

Exiting a loop

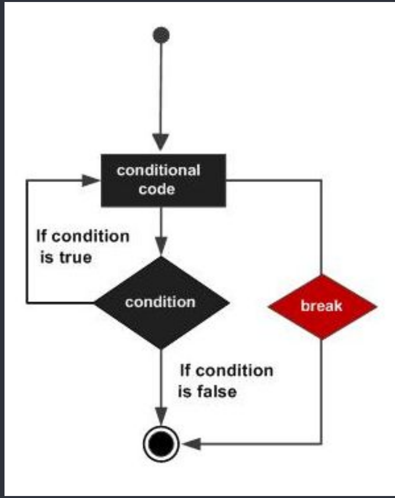
- C, C++, Python, Ruby, and C# have unconditional unlabelled exits
  <break>
- Java and Perl have unconditional labelled exits

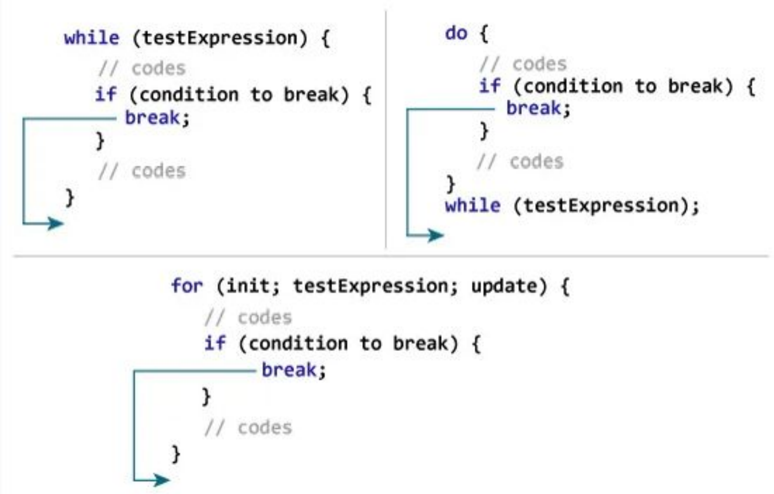
  <break -> Java>
  <last -> Perl>

}

User-Located Loop Control Mechanisms; {

Break statements





}

User-Located Loop Control Mechanisms; {

Java

```
var num = 0;
    for(var i = 0; i < 10; i++){
        for(var j = 0; j < 10 ; j++){
            if(i == 5 && j == 5){
                break;
            }
            num++;
        }
    }
console.log(num)
```

```
var num = 0;
outermost:
    for(var i = 0; i < 10; i++){
        for(var j = 0; j < 10 ; j++){
            if(i == 5 && j == 5){
                break outermost;
            }
            num++;
        }
    }
console.log(num)
```

}

User-Located Loop Control Mechanisms; {

C

```c
int main () {
    int a = 10;

    while( a < 20 ) {
        printf("value of a: %d\n", a);
        a++;
        if(a > 15) {
            break;
        }
    }
    return 0;
}
```

}

Is it possible for C to have an unconditional labelled exit?

- No, but we can achieve a similar effect using goto

User-Located Loop Control Mechanisms; {

C

```
int main(){
    for (int i=0; i<100; i++) {
        switch(i) {
            case 0: printf("just started\n"); break;
            case 10: printf("reached 10\n"); break;
            case 20: printf("reached 20; exiting loop.\n"); goto afterForLoop;
            case 30: printf("Will never be reached."); break;
        }
    }

afterForLoop:
    printf("first statement after for-loop.");

return 0;
}
```

}

User-Located Loop Control Mechanisms; {

```
fruits = ["apple", "banana",
"cherry"]

for x in fruits:

    print(x)

    if x == "banana":

        break
```

Output:

```
apple
banana
```

}

```
fruits = ["apple", "banana",
"cherry"]

for x in fruits:

  if x == "banana":

    break

  print(x)
```

Output:

```
apple
```

User-Located Loop Control Mechanisms; {

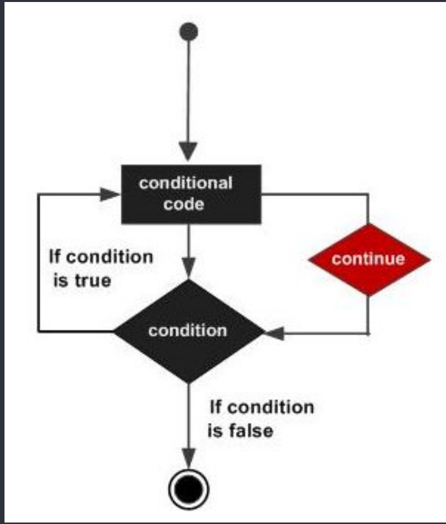
Unlabeled Control Statements (continue)

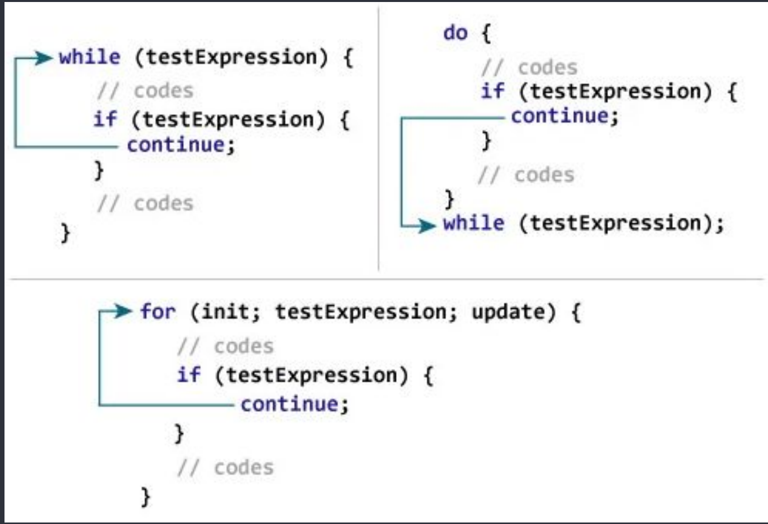
- available in C, C++, and Python
- transfers control to the control mechanism of the smallest enclosing loop
- This is not an exit but rather a way to skip the rest of the loop statements on the current iteration without terminating the loop construct

}

User-Located Loop Control Mechanisms; {

Continue statements





}

User-Located Loop Control Mechanisms; {

C

```c
int main(){
    int j;
    for (j = 0; j < 5; j++){
        if (j == 2)
            break;
        printf("%d ", j);
    }
    return 0;
}
```

```c
int main(){
    int j;
    for (j = 0; j < 5; j++){
        if (j == 2)
            continue;
        printf("%d ", j);
    }
    return 0;
}
```

}

User-Located Loop Control Mechanisms; {

C (Nested Loop)

```
#include<stdio.h>
int main(){
    int i, j;
    for(i=1; i<5; i++){
        for(j=1; j<5; j++){
            if(i%2==0 || j%2==0) continue;
            printf("%d\t%d\n",i,j);
        }
    }
    return 0;
}
```

Output:

```
1    3
3    3
3    1
3    3
```

}

User-Located Loop Control Mechanisms; {

Java

```java
public class Main {
  public static void main(String[] args) {
    int i = 0;
    while (i < 10) {
      if (i == 4) {
        i++;
        continue;
      }
      System.out.println(i);
      i++;
    }
  }
}
```

}

User-Located Loop Control Mechanisms; {

Java (Nested Loop)

```
int i = 1, j = 1;
while (i <= 3) {
    System.out.println("Outer Loop: " + i);
    while(j <= 3) {
        if(j == 2) {
            j++;
            continue;
        }
        System.out.println("Inner Loop: " + j);
        j++;
    }
    i++;
}
```

}

User-Located Loop Control Mechanisms; {

Java (Labelled Continue Statement)

```
first:
for (int i = 1; i < 6; ++i) {
    for (int j = 1; j < 5; ++j) {
        if (i == 3 || j == 2)
            continue first;
        System.out.println("i = " + i + "; j = " + j);
    }
}
```

}

User-Located Loop Control Mechanisms; {

Python

```
fruits = ["apple", "banana",
"cherry"]

for x in fruits:
     if x == "banana":
          continue
     print(x)
```

Output:

```
apple
cherry
```

}

User-Located Loop Control Mechanisms; {

Using both Break and Continue

```
int main() {
  int i=0;
  while(1){
      i++;
      if(i%2==1) continue;
      if(i>10) break;
      printf("%d \n", i);
  }
  return 0;
}
```

Output:

```
2
4
6
8
10
```

}

User-Located Loop Control Mechanisms;

{

Motivation for User-Located Loop Exits

- They fulfill a common need for goto statements using a highly restricted branch statement.
- The target of a goto can be many places in the program, both above and below the goto itself.
- However, the targets of user-located loop exits must be below the exit and can only follow immediately at the end of a compound statement.

}

Iteration Based on Data Structures;{

- uses a user-defined and function (iterator) to go through the structure’s elements
- each time it is called, the iterator returns an element from a particular data structure in some specific order
- terminates when the iterator fails to find more elements

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

}

Iteration Based on Data Structures;{

C

```
void insertSorted(char x, LIST *L){
      LIST temp, trav;
      temp = (LIST)malloc(sizeof(nodetype));
      for(trav = *L, *L != NULL && (strcmp(*L->elem, x) < 0); trav = trav->link){}
      if(temp != NULL){
            trav->link = temp;
            temp->elem = x;
            temp = temp->link;
            *L = temp;
      }else{
            temp->elem = x;
            temp->link = NULL;
      }
}
```

}

Iteration Based on Data Structures;{

- more important in OOP due to the use of abstract data types for data structures
- iterator is called at the beginning of each iteration
- each time it is called, the iterator returns an element from a particular data structure in some specific order
- terminates when the iterator fails to find more elements

}

Iteration Based on Data Structures;{

Java

Iterator

- object that can be used to loop through collections, like ArrayList and HashSet
- can modify a collection: removing an element or changing content of an item stored in the collection

For Each

- meant for traversing items in a collection
- can’t modify collection, as it will throw a ConcurrentModificationException

}

Iteration Based on Data Structures;{

Java (Iterator)

```java
import java.util.ArrayList;
import java.util.Iterator;

public class Main {
  public static void main(String[] args) {
    ArrayList<Integer> numbers = new ArrayList<Integer>();
    numbers.add(12);
    numbers.add(8);
    numbers.add(2);
    numbers.add(23);
    Iterator<Integer> it = numbers.iterator();
    while(it.hasNext()) {
      Integer i = it.next();
      if(i < 10) {
        it.remove();
      }
    }
    System.out.println(numbers);
  }
}
```

}

Iteration Based on Data Structures;{

Java (ListIterator)

```java
import java.util.ArrayList;
import java.util.ListIterator;

class Main {
  public static void main(String[] args) {
    ArrayList<Integer> numbers = new ArrayList<>();
    numbers.add(1);
    numbers.add(3);
    numbers.add(2);
    System.out.println("ArrayList: " + numbers);

    ListIterator<Integer> iterate = numbers.listIterator();
    System.out.println("Iterating over ArrayList:");
    while(iterate.hasNext()) {
      System.out.print(iterate.next() + ", ");
    }
  }
}
```

}

Iteration Based on Data Structures;{

Java 5.0

- An enhanced version simplifies iterating through the values in an array or objects in a collection that implements the Iterable interface

```
for (String myElement : myList) { . . . }
```

}

Iteration Based on Data Structures;{

Java 5.0 (For Each)

```java
public class Main {
  public static void main(String[] args) {
    String[] cars = {"Volvo", "BMW", "Ford", "Mazda"};
    for (String i : cars) {
      System.out.println(i);
    }
  }
}
```

Output:

```
Volvo
BMW
Ford
Mazda
```

}

Iteration Based on Data Structures;{

C# and F# (and the other .NET languages)

- Have predefined generic collections that have built-in iterators that are used implicitly with the foreach statement

```
List names = new List();
names.Add("Bob");
names.Add("Carol");
names.Add("Alice");
. . .
foreach (String name in names)
    Console.WriteLine(name);
```

}

Iteration Based on Data Structures;{

Ruby

- Block - a sequence of code, delimited by either braces or the do and end reserved words
- can be used with specially written methods to create many useful constructs, including iterators for data structures

}

Iteration Based on Data Structures;{

Ruby

```
upto(5) {|x| print x, " "}
```

output: 1 2 3 4 5

```
>> list = [2, 4, 6, 8]
=> [2, 4, 6, 8]
>> list.each {|value| puts value}
2
4
6
8
=> [2, 4, 6, 8]
```

}

Iteration Based on Data Structures;{

Yield (Python)

- acts like a return
- on the first call to traverse, yield returns the initial node of the structure. However, on the second call, it returns the second node
- any method that contains a yield statement is called a generator, because it generates data one element at a time

}

Iteration Based on Data Structures;{

Python

```
class MyStructure:
     # Other method definitions, including a constructor def
     traverse(self):
          # if there is another node:
          # set nod to next node
          # else:
          # return
          yield nod
```

}

Iteration Based on Data Structures;{

Python

```python
def simpleGeneratorFun():
    yield 1
    yield 2
    yield 3


for value in simpleGeneratorFun():
    print(value)
```

Output

```
1
2
3
```

}

Iteration Based on Data Structures;{

Python

```
def nextSquare():
    i = 1

    while True:
        yield i*i
        i += 1

for num in nextSquare():
    if num > 100:
        break
    print(num)
```

Output

```
1
4
9
16
25
36
49
64
81
100
```

}

8.4 {

[Unconditional Branching]

}

Unconditional Branching; {

- transfers execution control to a specified location in the program
- goto - most powerful statement for controlling the flow of execution
- has great flexibility, but makes it highly dangerous
- can make programs difficult to read; highly unreliable and costly

}

```c
#include <stdio.h>

int main(){
    int num=0;
    char choice;
    label1:
        printf("Enter a number greater than 5: ");
        scanf("%d", &num);

        if(num <=5){
            goto label1;
        }

    printf("You entered: %d\n", num);

    label2:
        printf("Do you want to enter another number? (y/n) ");
        scanf(" %c", &choice);

        if (choice == 'y'){
            goto label1;
        }else if (choice == 'n'){
            goto label3;
        }else{
            printf("Invalid choice, try again.\n");
            goto label2;
        }

    label3:
        printf("Goodbye\n");

    return 0;
}
```

Unconditional Branching; {

History

- Edsger Dijkstra gave the first widely read exposé on the dangers of the goto
- Readability is best when the execution order of statements in a program is nearly the same as the order in which they appear
- few languages designed without a goto: Java, Python, and Ruby
- Kernighan and Ritchie(1978) call the goto infinitely abusable
- Loop exit statements are camouflage goto statements

}

Unconditional Branching; {

- C# includes a goto used in the switch statement
- Loop exit statements are camouflage goto statements

}

8.5 {

[Guarded Command]

}

Guarded Command {

- Designed by Edsger Dijkstra
- Purpose: to support a new programming methodology that supported verification (correctness) during development
- Basis for two linguistic mechanisms for concurrent programming in CSP and Ada (Chapter 13)
- Used to define functions in Haskell (Chapter 15)

}

Selection Guarded Command {

- Form

closing reserved word is the opening reserved word spelled backward

```
if <Boolean Expression> → <statement>
[] <Boolean Expression> → <statement>
[] . . .
[] <Boolean Expression> → <statement>
fi
```

}

Selection Guarded Command {

- Form

Fatbars

- used separate the guarded clauses
- allow the clauses to be statement sequences

```
if <Boolean Expression> → <statement>
[] <Boolean Expression> → <statement>
[] . . .
[] <Boolean Expression> → <statement>
fi
```

}

Selection Guarded Command {

- Form

Guarded Command

- Boolean expression (a guard) and a statement or statement sequence

```
if <Boolean Expression> → <statement>
[] <Boolean Expression> → <statement>
[] . . .
[] <Boolean Expression> → <statement>
fi
```

}

Selection Guarded Command {

- Form

```
if <Boolean Expression> → <statement>
[] <Boolean Expression> → <statement>
[] . . .
[] <Boolean Expression> → <statement>
fi
```

}

Semantics: when construct is reached,

- Evaluate all Boolean expressions
- If more than one are true, choose one non-deterministically
- If none are true, it is a runtime error

Selection Guarded Command {

- Example

```
if i = 0 → sum := sum + i
[] i > j → sum := sum + j
[] j > i → sum := sum + k
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 runtime error occurs

}

Selection Guarded Command {

Traditional Programming Selector

```
if (x >= y)
    max = x;
else
    max = y;
```

- No practical difference
- This choice between the two statements complicates the formal analysis of the code and the correctness proof of it

Guarded Command

```
if x >= y → max := x
[] y >= x → max := y
fi
```

- Computes the desired result without overspecifying the solution

}

Loop Guarded Command {

- Form

```
do <Boolean Expression> → <statement>
[] <Boolean Expression> → <statement>
[] . . .
[] <Boolean Expression> → <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

Loop Guarded Command {

Example:

Given four integer variables, q1, q2, q3, and q4, rearrange the values of the four so that q1 <= q2 <= q3 <= q4.

- straightforward solution is to put the four values into an array,
- sort the array,
- and then assign the values from the array back into the scalar variables q1, q2, q3, and q4.

Guarded Command

```
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
```

}

Loop Guarded Command {

Guarded Command

```
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
```

}

Example:

```
q1= 9;
q2= 2;
q3= 5;
q4= 1;
```

<u>1st Iteration</u>

```
[] q1>q2 ⇒ 9>2
[] q2>q3 ⇒ 2>5
[] q3>q4 ⇒ 5>1
```

Choose non-deterministically

TRUE!

Loop Guarded Command {

Guarded Command

```
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
```

}

Example:

```
q1= 9;
q2= 2;
q3= 5;
q4= 1;
```

1st Iteration

```
[] q1>q2 ⇒ 9>2
[] q2>q3 ⇒ 2>5
[] q3>q4 ⇒ 5>1
```

Swap!

Loop Guarded Command {

Guarded Command

```
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
```

Example:

```
q1= 9;
q2= 2;
q3= 1;
q4= 5;
```

2nd Iteration

```
[] q1>q2 ⇒ 9>2
[] q2>q3 ⇒ 2>1
[] q3>q4 ⇒ 1>5
```

Choose non-deterministically

TRUE!

}

Loop Guarded Command {

Guarded Command

```
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
```

Example:

```
q1= 9;
q2= 2;
q3= 1;
q4= 5;
```

<u>2nd Iteration</u>

```
[] q1>q2 ⇒ 9>2
[] q2>q3 ⇒ 2>1
[] q3>q4 ⇒ 1>5
```

Swap!

}

Loop Guarded Command {

Guarded Command

```
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
```

Example:

```
q1= 2;
q2= 9;
q3= 1;
q4= 5;
```

3rd Iteration

```
[] q1>q2 ⇒ 2>9
[] q2>q3 ⇒ 9>1
[] q3>q4 ⇒ 1>5
```

True!

}

Loop Guarded Command {

Guarded Command

```
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
```

Example:

```
q1= 2;
q2= 9;
q3= 1;
q4= 5;
```

<u>3rd Iteration</u>

```
[] q1>q2 ⇒ 2>9
[] q2>q3 ⇒ 9>1
[] q3>q4 ⇒ 1>5
```

Swap!

}

Loop Guarded Command {

Guarded Command

```
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
```

}

Example:

q1= 2;

$\boxed{\text{q2= 1; q3= 9;}}$

q4= 5;

<u>4th Iteration</u>

[] $\boxed{q1>q2 \Rightarrow 2>1}$

[] $q2>q3 \Rightarrow 1>9$

[] $\boxed{q3>q4 \Rightarrow 9>5}$

Choose non-deterministically

True!

Loop Guarded Command {

Guarded Command

```
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
```

Example:

```
q1= 2;
q2= 1;
q3= 9;
q4= 5;
```

4th Iteration

```
[] q1>q2 ⇒ 2>1
[] q2>q3 ⇒ 1>9
[] q3>q4 ⇒ 9>5
```

Swap!

}

Loop Guarded Command {

Guarded Command

```
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
```

Example:

```
q1= 2;
q2= 1;
q3= 5;
q4= 9;
```

<u>5th Iteration</u>

```
[] q1>q2 ⇒ 2>1
[] q2>q3 ⇒ 1>5
[] q3>q4 ⇒ 5>9
```

True!

}

Loop Guarded Command {

Guarded Command

```
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
```

}

Example:

```
q1= 2;
q2= 1;
q3= 5;
q4= 9;
```

5th Iteration

```
[] q1>q2 ⇒ 2>1
[] q2>q3 ⇒ 1>5
[] q3>q4 ⇒ 5>9
```

Swap!

Loop Guarded Command {

Guarded Command

```
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
```

Example:

```
q1= 1;
q2= 2;
q3= 5;
q4= 9;
```

<u>6th Iteration</u>

```
[] q1>q2 ⇒ 1>2;
[] q2>q3 ⇒ 2>5;
[] q3>q4 ⇒ 5>9;
```

End of loop!

}

Guarded Commands: Rationale {

- Connection between control statements and program verification is intimate
- Verification is impossible with goto statements
- Verification is greatly simplified if (1) only logical loops and selections are used or (2) only guarded commands are used
- There is considerably increased complexity in the implementation of the guarded commands over their conventional deterministic counterparts.

}

8.6 {

[Conclusion]

}

Conclusion {

- Only sequence, selection, and pretest logical loops are absolutely required to express computations (Bohm and Jacopini, 1966)
- Choice of control statements beyond selection and logical pretest loops is a trade-off between language size and writability
- Disagreement on whether a language should include a goto
  - C++ and C# do
  - Java and Ruby do not

}

```
End {

        [Thank you!]

}
```