CHAPTER 6: CONTROL FLOW PART 2

CMPSC 460 – Principles of Programming Languages

Control Flow

- Basic paradigms for control flow:
  - Sequencing
  - Selection
  - Iteration
  - Procedural Abstraction
  - Recursion
  - Concurrency
  - Exception Handling and Speculation
  - Non-determinacy

Structured/Unstructured Flow

```
10 IF (X .GT. 0.000001) GO TO 20
   X = -X
11 Y = X*X - SIN(Y)/(X+1)
   IF (X .LT. 0.000001) GO TO 50
20 IF (X*Y .LT. 0.00001) GO TO 30
   X = X-Y-Y
30 X = X+Y
        ...
50 CONTINUE
   X = A
   Y = B-A + C*C
   GO TO 11
        ...
```

Structured Programming

- Goto statements were replaced by:
  - Selection statements
  - Iteration statements
  - Return statement
  - Break, exit statements
  - Continue, cycle, next
  - Exceptions

Structured Programming

- Structured Programming
  - Modular Development
  - Structured types
  - Descriptive identifier names
  - The emphasis on code documentation

Exercise

```
#include <iostream>
using namespace std;

int main() {
        for (int x = 0; x < 3; x++) {
        for (int y = 0; y < 3; y++) {
            cout << "(" << x << ";" << y << ") " << '\n';
            if (x + y >= 3)
                goto endloop;
        }
    }
endloop:
        return 0;
}
```

Nesting Selectors - Ambiguity

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

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

Nesting Selectors - Ambiguity

- Possible Solutions:
  1. All then and else statements should be something other than if
  2. All then and else statements should be compound
  3. The use of indentation
  4. The use of a terminating keyword
  5. The use of disambiguating rule

Nesting Selectors - Ambiguity

- Nested selection statements in Perl

```
$sum = 5;
$result = -5;
   if ($sum == 0)
 {
    if ($count == 0)
    {
      $result = 0;
     }
    else
    {
      $result = 1;
    }
  }
print($result)
```

Nesting Selectors - Ambiguity

- Nested selection statements in Python

```
sum = 5
result = -5
if sum == 0 :
   if count == 0 :
       result = 0
   else :
       result = 1
print(result)
```

Nesting Selectors - Ambiguity

- Nested selection statements in Ruby

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

Selection

- A *selection statement* provides the means of choosing between two or more paths of execution
- Two general categories:
  - Two-way selectors
  - Multiple-way selectors

Selection

- Selection
  - sequential if statements

```
if ... then ... else
if ... then ... elsif ... else
```

```
(cond
        ((C1)  (E1))
        ((C2)  (E2))
        ...
        ((Cn)  (En))
        ((T)   (Et))
)
```

Selection - shortcut

```
INT i := 3;
IF i < 9 THEN
    print(5)
ELSE print(6)
FI
```

Multiple-way selectors

```
ReadInt(i);
    IF i = 1 THEN
        WriteString("You typed 1.");
    ELSIF i IN {2, 7} THEN
        WriteString("You typed 2 or 7.");
    ELSIF i IN {3..5} THEN
        WriteString("You typed something between");
        WriteString(" 3 and 5.");
    ELSIF (i = 10) THEN
        WriteString("You typed 10.");
    ELSE
        WriteString("You typed something else.");
```

Case/Switch Statements

```
CASE i IN
   <action1>,
   <action2>,
   <action3>,
   <action4>,
OUT <action5>
ESAC

(i|action1,action2,action3,action4|action5)
```

Case/Switch Statements

```
INT i := 3;
REAL x := 4;

CASE i IN
   print(3),
   print(i:=(x>3.5|4|-2)),
   print(6)
OUT i := i+3
ESAC
```

Case/Switch Statements

- Design Issues:

1. Are label ranges or lists permitted
2. Is Fall through supported
3. How is this statement implemented
4. Is a default clause supported
5. What about unrepresented expression values
6. The type of the control expression
7. Is the order of cases important

Case/Switch Statements

```
#include <iostream>
using namespace std;

int main() {
        int total = 0, num =4;
        switch(num)
        {
                case 1:
                case 2: total = 5;
                case 3: total = 10;
                case 4: total = total + 3;
                case 8: total = total + 6;
                default: total = total + 4;
        }
        cout << total << endl;
        return 0;
}
```

Case/Switch Statements

```
score = 70
result = case score
  when 0..59 then "F"
  when 60..69 then "D"
  when 70..79 then "C"
  when 80..89 then "B"
  when 90..100 then "A"
  else "Invalid Score"
  end
puts result
```

Case/Switch Statements - Types

```
switch (month.toLowerCase())
    {
        case "january":
            monthNumber = 1;
            break;
        case "february":
            monthNumber = 2;
            break;
        case "march":
            monthNumber = 3;
            break;
        case "april":
            monthNumber = 4;
            break;
        default:
            monthNumber = 0;
            break;
    }
```

Case/Switch Statements

```
int month = 120;
switch (month) {
    default: monthText = "Invalid Month"; break;
    case 8:  monthText = "August"; break;
    case 1:  monthText = "January"; break;
    case 2:  monthText = "February"; break;
    case 12: monthText = "December"; break;
    case 4:  monthText = "April"; break;
    case 5:  monthText = "May"; break;
    case 6:  monthText = "June"; break;
    case 7:  monthText = "July";   break;
    case 10: monthText = "October"; break;
    case 9:  monthText = "September";break;
    case 3:  monthText = "March";  break;
    case 11: monthText = "November"; break;
}
```

Nested Case/Switch Statements

```
int count=0, x =2, sum = 0;

 switch(x)
 {
      case 1:
      case 2:
         switch(count)
         {
          case 0: cout <<"First Nested Switch Statement" <<endl;
         }
      default:
        switch(count)
        {
         case 0: cout <<"Second Nested Switch Statement" <<endl;
        }
 }
```

Case/Switch Statements - Implementation

```
CASE i OF
    1:      WriteString("You typed 1.");
|   2, 7:   WriteString("You typed 2 or 7.");
|   3..5:   WriteString("You typed something between");
            WriteString(" 3 and 5.");
|   10:     WriteString("You typed 10.");
    ELSE    WriteString("You typed something else.");
END;
```

Case/Switch Statements - Implementation

```
T:    &L1
      &L2
      &L3
      &L3
      &L3
      &L5
      &L2
      &L5
      &L5
      &L4
      goto L6
L1:   clause_A
      goto L7
L2:   clause_B
      goto L7
L3:   clause_C
      goto L7
      ...
L4:   clause_D
      goto L7
L5:   clause_E
      goto L7
L6:   r1 := ...
      if r1 < 1 goto L5
      if r1 > 10 goto L5
      r1 -:= 1
      r2 := T[r1]
      goto *r2
L7:
```

Case/Switch Statements

- How do simulate a switch statement in Python?

Case/Switch Statements

```
def zero():
    print ('Zero.')

def one():
    print ('one.')

def two():
    print ('two.')

def three():
    print ('three.')

def invalid():
       print ('Invalid Value.')
```

Case/Switch Statements

```
def numbers_to_strings(arg):
    switcher = {
        0: zero,
        1: one,
        2: two,
        3: three
    }
    return switcher.get(arg, invalid)()

print (numbers_to_strings(2))
```

Iteration

- General design issues for iteration control statements:
  1. Where is the control mechanism in the loop?
  2. How is iteration controlled?
     I. Enumeration controlled
     II. Logically controlled

Iteration – ALGOL 60

```
for i := 3, 7,
      11 step 1 until 16,
      i ÷ 2 while i >= 1,
      2 step i until 32 do
      print( i );
```

Iteration

- Pre-Test

  while *test* **do** *body*

- Post-Test

  do *body* **while** *test*

- How about Mid-Test?

  dowhiledo *body* **while** *(cond)* *body* **end**

Iteration – Enumeration Controlled

- Enumeration-controlled
  1. Type and scope of the loop variable
  2. Changes to bounds within loop
  3. Changes to loop variable within loop
  4. Can control enter or leave the loop
  5. Value after the loop

Iteration – Enumeration Controlled

- Ada

```
for var in [reverse] discrete_range loop
...
end loop;
```

- Example:

```
Count : Float := 1.35;
for Count in reverse 1 .. 10 loop
    Put_Line(natural'image(Count));
end loop;
```

Iteration – Enumeration Controlled

- Ada - Example:

```
with Ada.Text_IO; use Ada.Text_IO;
procedure Example  is
   Count : Float := 1.35;
   Sum : Integer := 0;
   x : Integer :=1;
   y : Integer :=10;
begin

    for Count in x .. y loop
        Sum := Sum + Count;
        x := 10;
        y:= 50;
    end loop;
   Put_Line(natural'image(sum));
end Example ;
```

Iteration – Enumeration Controlled

- C-based languages

```
for ([expr_1] ; [expr_2] ; [expr_3]) statement
```

- Design choices:
  - There is no explicit loop variable
  - Each expression can be single or multiple expressions
  - No need for a loop body
  - The first expression is evaluated once, but the other two are evaluated with each iteration

Iteration – Enumeration Controlled

```
#include <iostream>
using namespace std;

int main()
{
    int count1;
    float count2, sum;
    for (count1 = 0, count2 = 1.0; count1 <= 10 && count2 <=
        100.0; sum = ++count1 + count2, count2 *=2.5);
    cout << sum << endl;
    return 0;
}
```

Iteration – Enumeration Controlled

```
#include <iostream>
using namespace std;

int main()
{
    for (short i =0; i <= 32767; i++)
    cout << i<< " ";
    return 0;
}
```

Iteration – Enumeration Controlled

```
#include <iostream>
using namespace std;

int main()
{
   int count, x =10, sum = 0;

   for (count = 0; count <= x ; ++count )
   {
       sum += count;
       if (count ==5)
            x = 0;
   }
   cout <<"Count : " << count <<endl;
   cout <<"Sum : " <<  sum <<endl;

   return 0;
}
```

Iteration – Enumeration Controlled

```
#include <iostream>
using namespace std;

int main()
{
   int count, x =10, sum = 0;

   for (count = 0; count <= x ; ++count )
   {
       sum += count;
       if (count ==1)
            count = 10;
   }

   cout <<"Count : " << count <<endl;
   cout <<"Sum : " <<  sum <<endl;

   return 0;
}
```

Iteration – Enumeration Controlled

```
#include <iostream>
using namespace std;

int main()
{
   int count, sum = 0;

   for (count = 0; count <= 10 ; ++count )
        L1: sum += count;
   cout <<"Sum : " <<  sum <<endl;
   cout <<"Count : " << count <<endl;

   sum = 0;
   goto L1;
   cout << sum;

   return 0;
}
```

Iterators

- Perl

```
foreach $X (@arrayitem) { … }
```

- Python:

```
for count in range(5):
    print count
```

- Java:

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

- C#:

```
foreach (String name in names)
```

Iterators

- Ruby:

```
list= [2, 4, 6, 8, 10, 12]
list.each {|value| puts value}
list.each_with_index {|value, index| puts "#{index} #{value}"}

5.times { puts "Hello!"}

0.upto(10){ |y| print y, " "}
```

Recursion

- Recursion
  - sometimes more intuitive
  - other times less intuitive
  - implementation less efficient?
    - Optimizing compilers
      - The use of tail recursion

Recursion

```
int factorial(int n)
{
      if (n == 0)
            return 1;
      else
            return n * factorial(n - 1);
}
```

Recursion

```
factorial(6)
| factorial(5)
| |factorial(4)
| | |factorial(3)
| | | |factorial(2)
| | | | |factorial(1)
| | | | | |factorial(0)
| | | | | |1
| | | | |1
| | | |2
| | |6
| |24
| 120
720
```

Tail Recursion

- What is tail call?
  - No computation follows the call
- What is tail recursion?

Tail Recursion

- Tail recursion:

```
int factorialHelper(int n, int accumulator)
{
    if (n == 0)
        return accumulator;
    else
        return factorialHelper(n - 1, n * accumulator);
}

int factorial(int n)
{
    return factorialHelper(n, 1);
}
```

Recursion

```
(factHelper 6 1)
|(factHelper 5 6)
|(factHelper 4 30)
|(factHelper 3 120)
|(factHelper 2 360)
|(factHelper 1 720)
|(factHelper 0 720)
|720
720
```

Tail Recursion

- Iterative version:

```
int factorialHelper(int n, int accumulator)
{
  beginning:
    if (n == 0)
       return accumulator;
    else
   {
       accumulator *= n;
       n -= 1;
       goto beginning;
    }
  }
```

Tail Recursion – Exercise 1

```
(define lat?  (lambda (l)
    (cond
      ((null? l) #t)
      ((atom? (car l)) (lat? (cdr l)))
      (else #f))))
```

Tail Recursion – Exercise 2

```
(define insertR  (lambda (new old lat)
    (cond
      ((null? lat) '())
      ((eq? (car lat) old)
       (cons old (cons new (cdr lat))))
      (else
        (cons (car lat) (insertR new old (cdr lat)))))))
```

Non-Determinism

- Guarded commands:
  - Control statements that ensure correctness
  - Introduced:
    - Selection Guarded command
    - Loop Guarded command

Non-Determinism

- Selection Guarded commands:

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

Non-Determinism

- Selection Guarded commands:

```
if a >= b -> max := a
[] b >= a -> max := b
fi
```

Non-Determinism

- Selection Guarded commands:

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

Non-Determinism

```
int a;
a = 0;

if
:: (1 == 1) -> a = a + 1;
:: (2 == 2) -> a = a + 2;
:: (3 == 1) -> a = a + 7;
:: else -> a = 0;
fi
```

Non-Determinism

- Loop Guarded commands:

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

Non-Determinism

- Loop Guarded commands:

```
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
```

Non-Determinism

```
byte nr;
nr = 9;

do
:: nr++
:: nr--
:: break
do;
printf("nr: %d\n")
```

References

- Michael L. Scott, Programming Language Pragmatics, Morgan Kaufmann, 3rd edition, 2009.
- Robert W. Sebesta, Concepts of Programming Languages, Addison Wesley, 9th edition, 2009
- Robert W. Sebesta, Concepts of Programming Languages, Addison Wesley, 10th edition, 2012
- Terrence W. Pratt and Marvin V. Zelkowitz, Programming Languages: Design and Implementations, Prentice Hall, 4th edition, 2001.