Computer Science 433
Programming Languages
The College of Saint Rose
Fall 2014

Topic Notes: Control Structures

Nearly any useful program will need *control structures* – a mechanisms by which they can perform conditional execution and repetition. These quickly become familiar to introductory programmers.

In 1966, Böhm and Jacopini published "Flow Diagrams, Turing Machines, and Languages with only Two Formation Rules" in which they showed that all structured algorithms can be expressed using only three types of operations:

- sequence
- selection
- iteration

Further, they showed that all "flowchartable" code can be expressed using 2 control statements:

- two-way decision
- pretest loops

Anything more complex can be rewritten in terms of these basic building blocks.

However, most programming languages provide a wider variety of control structures.

Selection Statements

A *selection statement* allows the program to choose between/among two or more paths of execution. That is, which statement/block do we execute next?

These are often broken down into two categories: the two-way selector and the multi-way selector.

Two-way Selection

Two-way selection is provided by our familiar `if-then [ else ]` *conditional statements*:

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

Some things to consider about these statements:

- Is a "then" keyword included or required?
- Is some sort of parenthetical notation (*e.g.*, the curly braces in C/C++/Java, the `begin`/`end` in Pascal) required for compound clauses, or is indentation (*e.g.*, as in Python) used to determine the extent of the "then" and "else" parts?
- Generally, the control expression must evalulate to a Boolean, though C/C++ and Python allow an arithmetic expression.
- How is the *dangling else* problem handled?
  - we saw that the language grammar can handle this – Java uses this to match an `else` to the nearest previous `if` unless braces are used to specify something else
  - other languages eliminate the problem with an `end if` keyword (sometimes spelled `fi`)

Historical note: Fortran had an `IF` statement:

```
IF (I) GOTO 10,20,30
```

which performed the equivalent of the following (in a more familiar C/Java syntax):

```
if (i < 0) goto 10;
if (i == 0) goto 20;
goto 30;
```

It is easy to check these cases and peform the jump quickly with a 2's complement representation of `I`.

Multiple-way Selection

Here, we allow selection of one of many statements or blocks for execution.

This is provided in Java/C/C++ by the familiar `switch-case` construct as well as a chain of `if-else if-else` statements.

There are a number of variations on this construct and many issues to consider.

In C/C++/Java's `switch` statement:

- The control expression (the part in the parentheses after the `switch` keyword) must be integer types (no longer the case in Java 7 and up, which now allows `String` and enumerated types).

- Cases can be empty or overlapping – there is no implicit branch at the end of a selectable segment, necessitating the `break`.
- An optional `default` can be specified.

See Example:
`/home/cs433/examples/switch`

C# is similar except the "fall through" is disallowed and string constants can be used as selection values.

Ruby has a `case` statement that looks like this:

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

Multiple selectors may be implemented in a variety of ways:

- multiple conditional branches
- store case values in a table and use a linear search of the table
- a hash table of case values (typically for 10 or more case values)
- for small numbers of "small numbered" cases, could use an array indexed by the case values, values are the labels where to jump

We have also seen multiple selection with Scheme's `cond` function.

Iteration

Iteration is accomplished through loop constructs. These are often categorized as *counter-controlled loops* and *logic-controlled loops*.

Counter-Controlled Loops

A strict counting iterative statement is managed by a loop variable, and three values: the *initial*, *terminal* and *step* values.

We know well the `for` constructs that are used in the C-based languages (though the `for` construct is more general).

Fortran's `DO` statement is a more strictly a pure counting loop:

```
        DO 1000 I = 1, 99, 2
...
1000    CONTINUE
```

Some issues to consider with counting loops:

- legal types of loop variable?
- scope of loop variable?
- can the loop variable be modified in the body of the loop?
- if so, does it change the loop behavior?

In the C-based languages:

```
for ([e1]; [e2]; [e3]) statement
```

- all 3 expressions can be statements or statement sequences (with statements separated by commas, in which case its value is that of the last statement in the sequence)
- if `e2` is omitted, it is an infinite loop
- no explicit loop variable
- no restrictions on what can be modified within the loop
- `e1` is evaluated once, `e2`, `e3` are evaluated on each iteration
- a branch into the body of a `for` loop in C is legal!

  See Example:
  `/home/cs433/examples/gotoloops/gotofor.c`
- `break` terminates the loop immediately
- `continue` terminates the current iteration

C++ permits a Boolean control expression; Java and C# require it to be `boolean`.

Java, C++, C99 permit a variable declarations within `e1` with scope to the end of the loop, which C (before 99) does not.

An example in Ada:

On the web: For Loops in Ada Control, Wikibooks at
`http://en.wikibooks.org/wiki/Ada_Programming/Control#for_loop`

Here:

- the loop variable's type is a *discrete range*
- loop variable scope is limited to the loop
- loop variable cannot be changed in the loop, but the discrete range can; it does not affect loop control, since...
- the discrete range is evaluated just once
- cannot branch into the loop body

In Python:

On the web: "More Control Flow" in The Python Tutorial at
`http://docs.python.org/3/tutorial/controlflow.html`

- the loop variable is assigned each value in the specified range, once for each iteration
- there is an optional `else` clause that executes if the loop is executed to completion (no `break` caused an early exit)

Logic-Controlled Loops

These loops, of which the familiar `while` (*pre-test*) and `do-while` (*post-test*) loops in the C-based languages are examples, continue executing until a condition becomes false (or in the case of C, 0).

As with `for` loops, C and C++ `while` and `do-while` loops allow a branch into the loop body. Java has no mechanism that allows this, so it is not supported.

See Example:
`/home/cs433/examples/gotoloops/gotowhile.c`

Iterators

A special class of iteration is the traversal of the contents of a data structure, often based on an *iterator*.

We will look first at this in Java (5.0+) where arrays or any class that `implements Iterable`.

See Example:
`/home/cs433/examples/iterable`

Ruby has predefined iterator methods `times`, `each` and `upto`:

```
3.times {puts "Ruby is fun!"}
list.each {|value| puts value}
1.upto(5) {|x| print x, " "}
```

Perl allows iteration over an arbitraty list of values:

```
foreach $week (1..17,"divw","wc","div","conf","sb") {
  //  do something with $week here
}
```

Unconditional Branches

We have seen examples of unconditional branches like `break`, `continue`, and `goto`.

Early languages depended on the GOTO, structure languages emphasized higher-level control structures.

Edsger Dijkstra's famous letter to the editor of the *Communications of the ACM* sparked a great debate:

On the web: Letters to the editor: go to statement considered harmful at
`http://dl.acm.org/citation.cfm?doid=362929.362947`

Some modern languages do not include a general GOTO (*e.g.*, Java) but others do (*e.g.*, C, C++). Their use is generally quite rare.

Guarded Commands

Another control structure that you're less likely to have encountered in your programming experience is that of *guarded commands*. The idea is that a list of alternatives is presented and the program can choose, possibly randomly, among valid alternatives.

An example, from the text, using the notation proposed by Dijkstra when he proposed the idea in 1975.

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

When execution reaches this point, any of the statements after the `->` can be executed as long as the boolean expression preceding it, its *guard*, evaluates to true. In order to do this, all guards are evaluated (ideally concurrently), then one of the commands is chosen to execute, nondeterministically, from among those whose guards are true. Further, some guard must evaluate to true, or an error will occur.

In the above example, consider the possibilities, and what which statements could execute:

i=0, j=0 Only command 1 could execute

i=0, j=-1 Commands 1 or 2 could execute

i=0, j=1 Commands 1 or 3 could execute

i=1, j=0 Only command 2 could execute

i=1, j=1 No commands could execute, error

i=1, j=2 Only command 3 could execute

The main motivation for this was to ensure program correctness at run time, but the idea has come up again in the context of concurrency.

Another example, also from the text, shows how this can allow multiple options to be true when it does not matter which of two alternatives are equally valid:

```
if x >= y -> max := x
[] x <= y -> max := y
fi
```

We compute the max of two numbers, observing that it does not matter which of the two commands executes in the case where `x` is equal to `y`.

In Haskell, this idea can be used to define cases for a function (from Sebesta Ch. 15):

```
fact n
  | n == 0 = 1
  | n == 1 = 1
  | n > 1 = n * fact(n - 1)
```

Dijkstra also uses the concept of guarded commands for loop constructs. The loop continues so long as any guard is true, choosing nondeterministically from among the true guards to select a command to execute. Once all guards become false, the loop is complete.

The text shows how a loop with guarded commands can be used to express a sort routine for 4 values:

```
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
```

Each time through the loop. some value gets closer to its eventual location. Once all have arrived at their locations, all guards will be false, and the loop will terminate.